Principles of Operation
z/Architecture

Principles of Operation
## Contents

<table>
<thead>
<tr>
<th>Notices</th>
<th>xvi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trademarks</td>
<td>xvii</td>
</tr>
<tr>
<td><strong>Preface</strong></td>
<td>xix</td>
</tr>
<tr>
<td>Size and Number Notation</td>
<td>xx</td>
</tr>
<tr>
<td>Bytes, Characters, and Codes</td>
<td>xxi</td>
</tr>
<tr>
<td>Other Publications</td>
<td>xxi</td>
</tr>
<tr>
<td>Summary of Changes in Fourth Edition</td>
<td>xxii</td>
</tr>
<tr>
<td>Summary of Changes in Third Edition</td>
<td>xxiii</td>
</tr>
<tr>
<td>Summary of Changes in Second Edition</td>
<td>xxv</td>
</tr>
</tbody>
</table>

### Chapter 1. Introduction

<table>
<thead>
<tr>
<th>Highlights of Original z/Architecture</th>
<th>1-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Instructions for 64-Bit Integers</td>
<td>1-2</td>
</tr>
<tr>
<td>Other New General Instructions</td>
<td>1-2</td>
</tr>
<tr>
<td>Floating-Point Instructions</td>
<td>1-4</td>
</tr>
<tr>
<td>Control Instructions</td>
<td>1-4</td>
</tr>
<tr>
<td>Tridimensional Addressing</td>
<td>1-5</td>
</tr>
<tr>
<td>Modal Instructions</td>
<td>1-5</td>
</tr>
<tr>
<td>Effects on Bits 0-31 of a General Register</td>
<td>1-5</td>
</tr>
<tr>
<td>Input/Output</td>
<td>1-5</td>
</tr>
<tr>
<td>Additions to z/Architecture</td>
<td>1-7</td>
</tr>
<tr>
<td>Extended-Translation Facility 2</td>
<td>1-7</td>
</tr>
<tr>
<td>HFP Multiply-and-Add/Subtract Facility</td>
<td>1-7</td>
</tr>
<tr>
<td>Message-Security Assist</td>
<td>1-7</td>
</tr>
<tr>
<td>Long-Displacement Facility</td>
<td>1-8</td>
</tr>
<tr>
<td>Extended-I/O-Measurement-Block Facility</td>
<td>1-8</td>
</tr>
<tr>
<td>Extended-I/O-Measurement-Word Facility</td>
<td>1-8</td>
</tr>
<tr>
<td>Extended-Translation Facility 3</td>
<td>1-8</td>
</tr>
<tr>
<td>ASN-and-LX-Reuse Facility</td>
<td>1-9</td>
</tr>
<tr>
<td>List-Directed Initial Program Load</td>
<td>1-9</td>
</tr>
<tr>
<td>The ESA/390 Base</td>
<td>1-9</td>
</tr>
<tr>
<td>The ESA/370 and 370-XA Base</td>
<td>1-14</td>
</tr>
<tr>
<td>System Program</td>
<td>1-16</td>
</tr>
<tr>
<td>Compatibility</td>
<td>1-16</td>
</tr>
<tr>
<td>Compatibility among z/Architecture Systems</td>
<td>1-16</td>
</tr>
<tr>
<td>Compatibility between z/Architecture and ESA/390</td>
<td>1-17</td>
</tr>
<tr>
<td>Control-Program Compatibility</td>
<td>1-17</td>
</tr>
<tr>
<td>Problem-State Compatibility</td>
<td>1-17</td>
</tr>
<tr>
<td>Availability</td>
<td>1-17</td>
</tr>
</tbody>
</table>

### Chapter 2. Organization

<table>
<thead>
<tr>
<th>Main Storage</th>
<th>2-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expanded Storage</td>
<td>2-2</td>
</tr>
<tr>
<td>CPU</td>
<td>2-2</td>
</tr>
<tr>
<td>PSW</td>
<td>2-3</td>
</tr>
<tr>
<td>General Registers</td>
<td>2-3</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>2-3</td>
</tr>
<tr>
<td>Floating-Point-Control Register</td>
<td>2-4</td>
</tr>
<tr>
<td>Control Registers</td>
<td>2-4</td>
</tr>
<tr>
<td>Access Registers</td>
<td>2-4</td>
</tr>
<tr>
<td>Cryptographic Facility</td>
<td>2-6</td>
</tr>
<tr>
<td>External Time Reference</td>
<td>2-6</td>
</tr>
<tr>
<td>I/O Channel Subsystem</td>
<td>2-6</td>
</tr>
<tr>
<td>Channel Paths</td>
<td>2-6</td>
</tr>
<tr>
<td>I/O Devices and Control Units</td>
<td>2-7</td>
</tr>
<tr>
<td>Operator Facilities</td>
<td>2-7</td>
</tr>
</tbody>
</table>

### Chapter 3. Storage

<table>
<thead>
<tr>
<th>Storage Addressing</th>
<th>3-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Formats</td>
<td>3-2</td>
</tr>
<tr>
<td>Integral Boundaries</td>
<td>3-3</td>
</tr>
<tr>
<td>Address Types and Formats</td>
<td>3-3</td>
</tr>
<tr>
<td>Address Types</td>
<td>3-3</td>
</tr>
<tr>
<td>Absolute Address</td>
<td>3-3</td>
</tr>
<tr>
<td>Real Address</td>
<td>3-4</td>
</tr>
<tr>
<td>Virtual Address</td>
<td>3-4</td>
</tr>
<tr>
<td>Primary Virtual Address</td>
<td>3-4</td>
</tr>
<tr>
<td>Secondary Virtual Address</td>
<td>3-4</td>
</tr>
<tr>
<td>AR-Specified Virtual Address</td>
<td>3-5</td>
</tr>
<tr>
<td>Home Virtual Address</td>
<td>3-5</td>
</tr>
<tr>
<td>Logical Address</td>
<td>3-5</td>
</tr>
<tr>
<td>Instruction Address</td>
<td>3-5</td>
</tr>
<tr>
<td>Effective Address</td>
<td>3-5</td>
</tr>
<tr>
<td>Address Size and Wraparound</td>
<td>3-5</td>
</tr>
<tr>
<td>Address Wraparound</td>
<td>3-6</td>
</tr>
<tr>
<td>Storage Key</td>
<td>3-8</td>
</tr>
<tr>
<td>Protection</td>
<td>3-9</td>
</tr>
<tr>
<td>Key-Controlled Protection</td>
<td>3-9</td>
</tr>
<tr>
<td>Storage-Protection-Override Control</td>
<td>3-10</td>
</tr>
<tr>
<td>Fetch-Protection-Override Control</td>
<td>3-11</td>
</tr>
<tr>
<td>Access-List-Controlled Protection</td>
<td>3-11</td>
</tr>
<tr>
<td>Page Protection</td>
<td>3-11</td>
</tr>
<tr>
<td>Low-Address Protection</td>
<td>3-12</td>
</tr>
<tr>
<td>Suppression on Protection</td>
<td>3-12</td>
</tr>
<tr>
<td>Reference Recording</td>
<td>3-14</td>
</tr>
<tr>
<td>Change Recording</td>
<td>3-14</td>
</tr>
<tr>
<td>Prefacing</td>
<td>3-15</td>
</tr>
<tr>
<td>Address Spaces</td>
<td>3-16</td>
</tr>
<tr>
<td>Changing to Different Address Spaces</td>
<td>3-17</td>
</tr>
<tr>
<td>Address-Space Number</td>
<td>3-17</td>
</tr>
<tr>
<td>ASN-Second-Table-Entry Sequence</td>
<td>3-18</td>
</tr>
<tr>
<td>Number</td>
<td>3-18</td>
</tr>
<tr>
<td>ASN-Second-Table-Entry Instance</td>
<td>3-19</td>
</tr>
<tr>
<td>Number and ASN Reuse</td>
<td>3-19</td>
</tr>
<tr>
<td>ASN Translation</td>
<td>3-24</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1990-2004
<table>
<thead>
<tr>
<th>Subspace Groups</th>
<th>5-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subspace-Group Tables</td>
<td>5-64</td>
</tr>
<tr>
<td>Subspace-Group Dispatchable-Unit</td>
<td></td>
</tr>
<tr>
<td>Control Table</td>
<td>5-64</td>
</tr>
<tr>
<td>Subspace-Group ASN-Second-Table</td>
<td></td>
</tr>
<tr>
<td>Entries</td>
<td>5-66</td>
</tr>
<tr>
<td>Subspace-Replacement Operations</td>
<td>5-68</td>
</tr>
<tr>
<td>Linkage-Stack Introduction</td>
<td>5-69</td>
</tr>
<tr>
<td>Summary</td>
<td>5-69</td>
</tr>
<tr>
<td>Linkage-Stack Functions</td>
<td>5-70</td>
</tr>
<tr>
<td>Transferring Program Control</td>
<td>5-70</td>
</tr>
<tr>
<td>Branching Using the Linkage Stack</td>
<td>5-72</td>
</tr>
<tr>
<td>Adding and Retrieving Information</td>
<td>5-73</td>
</tr>
<tr>
<td>Testing Authorization</td>
<td>5-73</td>
</tr>
<tr>
<td>Program-Problem Analysis</td>
<td>5-74</td>
</tr>
<tr>
<td>Linkage-Stack Entry-Table Entries</td>
<td>5-74</td>
</tr>
<tr>
<td>Linkage-Stack Operations</td>
<td>5-75</td>
</tr>
<tr>
<td>Linkage-Stack-Operations Control</td>
<td>5-77</td>
</tr>
<tr>
<td>Control Register 0</td>
<td>5-77</td>
</tr>
<tr>
<td>Control Register 15</td>
<td>5-77</td>
</tr>
<tr>
<td>Linkage Stack</td>
<td>5-78</td>
</tr>
<tr>
<td>Entry Descriptors</td>
<td>5-78</td>
</tr>
<tr>
<td>Header Entries</td>
<td>5-79</td>
</tr>
<tr>
<td>Trailer Entries</td>
<td>5-80</td>
</tr>
<tr>
<td>State Entries</td>
<td>5-80</td>
</tr>
<tr>
<td>Stacking Process</td>
<td>5-83</td>
</tr>
<tr>
<td>Locating Space for a New Entry</td>
<td>5-83</td>
</tr>
<tr>
<td>Forming the New Entry</td>
<td>5-84</td>
</tr>
<tr>
<td>Updating the Current Entry</td>
<td>5-85</td>
</tr>
<tr>
<td>Updating Control Register 15</td>
<td>5-86</td>
</tr>
<tr>
<td>Recognition of Exceptions during the Stacking Process</td>
<td>5-86</td>
</tr>
<tr>
<td>Unstacking Process</td>
<td>5-86</td>
</tr>
<tr>
<td>Locating the Current Entry and Processing a Header Entry</td>
<td>5-86</td>
</tr>
<tr>
<td>Checking for a State Entry</td>
<td>5-87</td>
</tr>
<tr>
<td>Restoring Information</td>
<td>5-87</td>
</tr>
<tr>
<td>Updating the Preceding Entry</td>
<td>5-88</td>
</tr>
<tr>
<td>Updating Control Register 15</td>
<td>5-88</td>
</tr>
<tr>
<td>Recognition of Exceptions during the Unstacking Process</td>
<td>5-88</td>
</tr>
<tr>
<td>Sequence of Storage References</td>
<td>5-88</td>
</tr>
<tr>
<td>Conceptual Sequence</td>
<td>5-89</td>
</tr>
<tr>
<td>Overlapped Operation of Instruction Execution</td>
<td>5-89</td>
</tr>
<tr>
<td>Interlocks for Virtual-Storage References</td>
<td>5-90</td>
</tr>
<tr>
<td>Interlocks between Instructions</td>
<td>5-90</td>
</tr>
<tr>
<td>Interlocks within a Single Instruction</td>
<td>5-91</td>
</tr>
<tr>
<td>Instruction Fetching</td>
<td>5-93</td>
</tr>
<tr>
<td>ART-Table and DAT-Table Fetches</td>
<td>5-94</td>
</tr>
<tr>
<td>Storage-Key Accesses</td>
<td>5-95</td>
</tr>
<tr>
<td>Storage-Operand References</td>
<td>5-96</td>
</tr>
<tr>
<td>Storage-Operand Fetch References</td>
<td>5-96</td>
</tr>
<tr>
<td>Storage-Operand Store References</td>
<td>5-96</td>
</tr>
<tr>
<td>Storage-Operand Update References</td>
<td>5-97</td>
</tr>
<tr>
<td>Storage-Operand Consistency</td>
<td>5-98</td>
</tr>
<tr>
<td>Single-Access References</td>
<td>5-98</td>
</tr>
<tr>
<td>Multiple-Access References</td>
<td>5-98</td>
</tr>
<tr>
<td>Block- Concurrent References</td>
<td>5-99</td>
</tr>
<tr>
<td>Consistency Specification</td>
<td>5-99</td>
</tr>
<tr>
<td>Relation between Operands</td>
<td>5-101</td>
</tr>
<tr>
<td>Other Storage References</td>
<td>5-101</td>
</tr>
<tr>
<td>Relation between Storage-Key Accesses</td>
<td>5-101</td>
</tr>
<tr>
<td>Serialization</td>
<td>5-102</td>
</tr>
<tr>
<td>CPU Serialization</td>
<td>5-102</td>
</tr>
<tr>
<td>Channel-Program Serialization</td>
<td>5-103</td>
</tr>
</tbody>
</table>

Chapter 6. Interruptions

Interruption Action

Enabling and Disabling
Handling of Floating Interruption Conditions
Instruction-Length Code
Zero ILC
ILC on Instruction-Fetching Exceptions
Exceptions Associated with the PSW
Early Exception Recognition
Late Exception Recognition
External Interruption
Clock Comparator
CPU Timer
Emergency Signal
ETR
External Call
Interrupt Key
Malfunction Alert
Service Signal
I/O Interruption
Machine-Check Interruption
Program Interruption
Data-Exception Code (DXC)
Priority of Program Interruptions for Data Exceptions
Program-Interruption Conditions
Addressing Exception
AFX-Translation Exception
ALEN-Translation Exception
ALE-Sequence Exception
ALET-Specification Exception
ASCE-Type Exception

vi z/Architecture Principles of Operation
Chapter 7. General Instructions

Data Format
Binary-Integer Representation
Binary Arithmetic
Signed Binary Arithmetic
Addition and Subtraction
Fixed-Point Overflow
Signed Binary Arithmetic
Signed and Logical Comparison
Instructions
ADD
ADD HALFWORD
ADD HALFWORD IMMEDIATE
ADD LOGICAL
ADD LOGICAL WITH CARRY
AND
AND IMMEDIATE
BRANCH AND LINK
BRANCH AND SAVE
BRANCH AND SAVE AND SET MODE
BRANCH AND SET MODE
BRANCH ON CONDITION
BRANCH ON COUNT
BRANCH ON INDEX HIGH
BRANCH ON INDEX LOW OR EQUAL
BRANCH RELATIVE AND SAVE
BRANCH RELATIVE AND SAVE LONG
BRANCH RELATIVE ON CONDITION
BRANCH RELATIVE ON CONDITION LONG
BRANCH RELATIVE ON COUNT
BRANCH RELATIVE ON INDEX HIGH
BRANCH RELATIVE ON INDEX LOW OR EQUAL
CHECKSUM
CIPHER MESSAGE (KM)
CIPHER MESSAGE WITH CHAINING (KMC)
COMPARE
COMPARE AND FORM CODEWORD
COMPARE AND SWAP
COMPARE DOUBLE AND SWAP
COMPARE HALFWORD
COMPARE HALFWORD IMMEDIATE
COMPARE LOGICAL
COMPARE LOGICAL CHARACTERS UNDER MASK
COMPARE LOGICAL LONG
COMPARE LOGICAL LONG EXTENDED
COMPARE LOGICAL LONG UNICODE
COMPARE LOGICAL STRING

Collective Program-Interruption Names
Revised Interruption
Supervisor-Call Interruption
Priority of Interruptions
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARE UNTIL SUBSTRING EQUAL</td>
<td>7-68</td>
</tr>
<tr>
<td>COMPRESSION CALL</td>
<td>7-72</td>
</tr>
<tr>
<td>COMPUTE INTERMEDIATE MESSAGE DIGEST (KIMD)</td>
<td>7-84</td>
</tr>
<tr>
<td>COMPUTE LAST MESSAGE DIGEST (KLMD)</td>
<td>7-84</td>
</tr>
<tr>
<td>COMPUTE MESSAGE AUTHENTICATION CODE (KMAC)</td>
<td>7-91</td>
</tr>
<tr>
<td>CONVERT TO BINARY</td>
<td>7-97</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL</td>
<td>7-98</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-32</td>
<td>7-98</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-8</td>
<td>7-101</td>
</tr>
<tr>
<td>CONVERT UTF-32 TO UTF-16</td>
<td>7-104</td>
</tr>
<tr>
<td>CONVERT UTF-32 TO UTF-8</td>
<td>7-107</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UTF-16</td>
<td>7-110</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UNICODE</td>
<td>7-110</td>
</tr>
<tr>
<td>COPY ACCESS</td>
<td>7-116</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>7-116</td>
</tr>
<tr>
<td>DIVIDE LOGICAL</td>
<td>7-117</td>
</tr>
<tr>
<td>DIVIDE SINGLE</td>
<td>7-117</td>
</tr>
<tr>
<td>EXCLUSIVE OR</td>
<td>7-118</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>7-119</td>
</tr>
<tr>
<td>EXTRACT ACCESS</td>
<td>7-120</td>
</tr>
<tr>
<td>EXTRACT PSW</td>
<td>7-120</td>
</tr>
<tr>
<td>INSERT CHARACTER</td>
<td>7-121</td>
</tr>
<tr>
<td>INSERT CHARACTERS UNDER MASK</td>
<td>7-121</td>
</tr>
<tr>
<td>INSERT IMMEDIATE</td>
<td>7-122</td>
</tr>
<tr>
<td>INSERT PROGRAM MASK</td>
<td>7-122</td>
</tr>
<tr>
<td>LOAD</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD ACCESS MULTIPLE</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD ADDRESS</td>
<td>7-124</td>
</tr>
<tr>
<td>LOAD ADDRESS EXTENDED</td>
<td>7-124</td>
</tr>
<tr>
<td>LOAD ADDRESS RELATIVE LONG</td>
<td>7-125</td>
</tr>
<tr>
<td>LOAD AND TEST</td>
<td>7-126</td>
</tr>
<tr>
<td>LOAD BYTE</td>
<td>7-126</td>
</tr>
<tr>
<td>LOAD COMPLEMENT</td>
<td>7-126</td>
</tr>
<tr>
<td>LOAD HALFWORD</td>
<td>7-127</td>
</tr>
<tr>
<td>LOAD HALFWORD IMMEDIATE</td>
<td>7-127</td>
</tr>
<tr>
<td>LOAD LOGICAL</td>
<td>7-128</td>
</tr>
<tr>
<td>LOAD LOGICAL CHARACTER</td>
<td>7-128</td>
</tr>
<tr>
<td>LOAD LOGICAL HALFWORD</td>
<td>7-128</td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE</td>
<td>7-128</td>
</tr>
<tr>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>7-129</td>
</tr>
<tr>
<td>LOAD MULTIPLE</td>
<td>7-129</td>
</tr>
<tr>
<td>LOAD MULTIPLE DISJOINT</td>
<td>7-130</td>
</tr>
<tr>
<td>LOAD MULTIPLE HIGH</td>
<td>7-130</td>
</tr>
<tr>
<td>LOAD NEGATIVE</td>
<td>7-130</td>
</tr>
<tr>
<td>LOAD PAIR FROM QUADWORD</td>
<td>7-131</td>
</tr>
<tr>
<td>LOAD POSITIVE</td>
<td>7-131</td>
</tr>
<tr>
<td>LOAD REVERSED</td>
<td>7-132</td>
</tr>
<tr>
<td>MONITOR CALL</td>
<td>7-133</td>
</tr>
<tr>
<td>MOVE</td>
<td>7-133</td>
</tr>
<tr>
<td>MOVE INVERSE</td>
<td>7-134</td>
</tr>
<tr>
<td>MOVE LONG</td>
<td>7-134</td>
</tr>
<tr>
<td>MOVE LONG EXTENDED</td>
<td>7-139</td>
</tr>
<tr>
<td>MOVE LONG UNICODE</td>
<td>7-142</td>
</tr>
<tr>
<td>MOVE NUMERICS</td>
<td>7-146</td>
</tr>
<tr>
<td>MOVE STRING</td>
<td>7-146</td>
</tr>
<tr>
<td>MOVE WITH OFFSET</td>
<td>7-147</td>
</tr>
<tr>
<td>MOVE ZONES</td>
<td>7-148</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>7-148</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL</td>
<td>7-149</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL IMMEDIATE</td>
<td>7-149</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL IMMEDIATE with BORROW</td>
<td>7-150</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL IMMEDIATE with BORROW</td>
<td>7-150</td>
</tr>
<tr>
<td>OR</td>
<td>7-151</td>
</tr>
<tr>
<td>OR IMMEDIATE</td>
<td>7-152</td>
</tr>
<tr>
<td>PACK</td>
<td>7-153</td>
</tr>
<tr>
<td>PACK ASCII</td>
<td>7-154</td>
</tr>
<tr>
<td>PACK UNICHAR</td>
<td>7-155</td>
</tr>
<tr>
<td>PERFORM LOCKED OPERATION</td>
<td>7-156</td>
</tr>
<tr>
<td>ROTATE LEFT SINGLE LOGICAL</td>
<td>7-171</td>
</tr>
<tr>
<td>SEARCH STRING</td>
<td>7-172</td>
</tr>
<tr>
<td>SEARCH STRING UNICHAR</td>
<td>7-173</td>
</tr>
<tr>
<td>SET ACCESS</td>
<td>7-174</td>
</tr>
<tr>
<td>SET ADDRESSING MODE</td>
<td>7-175</td>
</tr>
<tr>
<td>SET PROGRAM MASK</td>
<td>7-175</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE</td>
<td>7-176</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE LOGICAL</td>
<td>7-176</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE</td>
<td>7-177</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL</td>
<td>7-178</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL with BORROW</td>
<td>7-178</td>
</tr>
<tr>
<td>SHIFT RIGHT DOUBLE</td>
<td>7-178</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE</td>
<td>7-179</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE LOGICAL</td>
<td>7-180</td>
</tr>
<tr>
<td>STORE</td>
<td>7-180</td>
</tr>
<tr>
<td>STORE ACCESS MULTIPLE</td>
<td>7-181</td>
</tr>
<tr>
<td>STORE CHARACTER</td>
<td>7-181</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK</td>
<td>7-181</td>
</tr>
<tr>
<td>STORE CLOCK</td>
<td>7-182</td>
</tr>
<tr>
<td>STORE CLOCK EXTENDED</td>
<td>7-183</td>
</tr>
<tr>
<td>STORE HALFWORD</td>
<td>7-185</td>
</tr>
<tr>
<td>STORE HALFWORD IMMEDIATE</td>
<td>7-185</td>
</tr>
<tr>
<td>STORE MULTIPLE</td>
<td>7-185</td>
</tr>
<tr>
<td>STORE PAIR TO QUADWORD</td>
<td>7-186</td>
</tr>
<tr>
<td>STORE REVERSED</td>
<td>7-187</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>7-187</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL</td>
<td>7-188</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL with BORROW</td>
<td>7-189</td>
</tr>
<tr>
<td>SUPERVISOR CALL</td>
<td>7-190</td>
</tr>
<tr>
<td>TEST ADDRESSING MODE</td>
<td>7-190</td>
</tr>
<tr>
<td>TEST AND SET</td>
<td>7-191</td>
</tr>
<tr>
<td>TEST UNDER MASK (TEST UNDER MASK HIGH, TEST UNDER MASK LOW)</td>
<td>7-191</td>
</tr>
</tbody>
</table>
TRANSLATE .......................... 7-192
TRANSLATE AND TEST ............... 7-193
TRANSLATE AND TEST REVERSE .... 7-194
TRANSLATE EXTENDED ............... 7-195
TRANSLATE ONE TO ONE ............. 7-198
TRANSLATE ONE TO TWO ............. 7-198
TRANSLATE TWO TO ONE ............. 7-198
TRANSLATE TWO TO TWO ............. 7-198
UNPACK .............................. 7-202
UNPACK ASCII ....................... 7-203
UNPACK UNICODE .................... 7-204
UPDATE TREE ......................... 7-205

Chapter 8. Decimal Instructions ..... 8-1
Decimal-Number Formats .............. 8-1
 Zoned Format ........................ 8-1
 Packed Format ....................... 8-1
 Decimal Codes ....................... 8-2
 Decimal Operations .................. 8-2
 Decimal-Arithmetic Instructions ..... 8-2
 Editing Instructions .................. 8-3
 Execution of Decimal Instructions . 8-3
 Other Instructions for Decimal Operands . 8-3
 Decimal-Operand Data Exception .... 8-4
Instructions .......................... 8-4
 ADD DECIMAL ........................ 8-5
 COMPARE DECIMAL .................... 8-6
 DIVIDE DECIMAL ...................... 8-6
 EDIT ................................ 8-7
 EDIT AND MARK ....................... 8-9
 MULTIPLY DECIMAL ................... 8-11
 SHIFT AND ROUND DECIMAL .......... 8-11
 SUBTRACT DECIMAL ................... 8-12
 TEST DECIMAL ........................ 8-13
 ZERO AND ADD ....................... 8-13

Chapter 9. Floating-Point Overview and
Support Instructions .................. 9-1
Registers And Controls ............... 9-2
 Floating-Point Registers ............. 9-2
 Additional Floating-Point (AFP)
 Registers ............................ 9-2
 Valid Floating-Point-Register
 Designations ........................ 9-2
 Floating-Point-Control (FPC) Register . 9-2
 AFP-Register-Control Bit ............. 9-2
 Explicit Rounding Methods ........... 9-3
 Summary of Rounding Action ......... 9-3
Comparison of BFP and HFP Number
 Representations ....................... 9-4
 BFP and HFP Number Ranges .......... 9-4
 Equivalent BFP and HFP Number
 Representations ....................... 9-4
Instructions .......................... 9-6
 CONVERT BFP TO HFP .................. 9-8
 CONVERT HFP TO BFP ................. 9-9
 LOAD ................................ 9-10
 LOAD ZERO .......................... 9-11
 STORE ............................... 9-11
 Summary of All Floating-Point Instructions . 9-12

Chapter 10. Control Instructions ..... 10-1
BRANCH AND SET AUTHORITY .......... 10-6
BRANCH AND STACK ................... 10-10
BRANCH IN SUBSPACE GROUP .......... 10-13
COMPARE AND SWAP AND PURGE ....... 10-18
DIAGNOSE ............................. 10-19
EXTRACT AND SET EXTENDED
 AUTHORITY ............................ 10-21
EXTRACT PRIMARY ASN ............... 10-21
EXTRACT PRIMARY ASN AND
 INSTANCE ............................. 10-21
EXTRACT SECONDARY ASN ............ 10-22
EXTRACT SECONDARY ASN AND
 INSTANCE ............................. 10-22
EXTRACT STACKED REGISTERS ........ 10-22
EXTRACT STACKED STATE .............. 10-24
INSERT ADDRESS SPACE CONTROL .... 10-27
INSERT PSW KEY ...................... 10-28
INSERT STORAGE KEY EXTENDED ..... 10-28
INSERT VIRTUAL STORAGE KEY ...... 10-29
INVALIDATE DAT TABLE ENTRY ...... 10-30
INVALIDATE PAGE TABLE ENTRY ...... 10-34
LOAD ADDRESS SPACE
 PARAMETERS .......................... 10-36
LOAD CONTROL ....................... 10-45
LOAD PSW ............................ 10-45
LOAD PSW EXTENDED ................. 10-46
LOAD REAL ADDRESS .................. 10-47
LOAD USING REAL ADDRESS .......... 10-52
MODIFY STACKED STATE .............. 10-52
MOVE PAGE ........................... 10-54
MOVE TO PRIMARY .................... 10-56
MOVE TO SECONDARY ................. 10-56
MOVE WITH DESTINATION KEY ....... 10-58
MOVE WITH KEY ...................... 10-59
MOVE WITH SOURCE KEY .............. 10-60
PAGE IN ................................ 10-61
PAGE OUT ............................ 10-62
PROGRAM CALL ....................... 10-63
PROGRAM RETURN ..................... 10-76
PROGRAM TRANSFER ................... 10-81
PROGRAM TRANSFER WITH
 INSTANCE ............................. 10-81
PURGE ALB .......................... 10-91
PURGE TLB ........................... 10-91
RESET REFERENCE BIT EXTENDED ..... 10-91
RESUME PROGRAM ..................... 10-92
Chapter 11. Machine-Check Handling . 11-1
Machine-Check Detection . 11-2
Correction of Machine Malfunctions . 11-2
Error Checking and Correction . 11-2
CPU Retry . 11-2
Effects of CPU Retry . 11-3
Checkpoint Synchronization . 11-3
Handling of Machine Checks during
Checkpoint Synchronization . 11-3
Checkpoint-Synchronization Operations . 11-3
Checkpoint-Synchronization Action . 11-4
Channel-Subsystem Recovery . 11-4
Unit Deletion . 11-4
Handling of Machine Checks . 11-5
Validation . 11-5
Invalid CBC in Storage . 11-6
Programmed Validation of Storage . 11-7
Invalid CBC in Storage Keys . 11-7
Invalid CBC in Registers . 11-10
Check-Stop State . 11-11
System Check Stop . 11-11
Machine-Check Interruption . 11-11
Exigent Conditions . 11-11
Repressible Conditions . 11-12
Interruption Action . 11-12
Point of Interruption . 11-14
Machine-Check-Interruption Code . 11-15
Subclass . 11-16
System Damage . 11-16
Instruction-Processing Damage . 11-16
System Recovery . 11-16
Timing-Facility Damage . 11-16
External Damage . 11-17
Degradation . 11-17
Warning . 11-17
Channel Report Pending . 11-17
Service-Processor Damage . 11-18
Channel-Subsystem Damage . 11-18
Subclass Modifiers . 11-18
Backed Up . 11-18
Delayed Access Exception . 11-18
Ancillary Report . 11-18
Synchronous
Machine-Check-Interruption Conditions . 11-18
Processing Backup . 11-18
Processing Damage . 11-19
Storage Errors . 11-19
Storage Error Uncorrected . 11-19
Storage Error Corrected . 11-20
Storage Degradation . 11-20
Indirect Storage Error . 11-20
Machine-Check Interruption-Code
Validity Bits . 11-21
PSW-MWP Validity . 11-21
PSW Mask and Key Validity . 11-21
PSW Program-Mask and
Condition-Code Validity . 11-21
PSW-Instruction-Address Validity . 11-21
Failing-Storage-Address Validity . 11-21
External-Damage-Code Validity . 11-21
Floating-Point-Register Validity . 11-21
General-Register Validity . 11-21
Control-Register Validity . 11-21
Storage Logical Validity . 11-22
Access-Register Validity . 11-22
TOD-Programmable-Register Validity . 11-22
Floating-Point-Control-Register
Validity . 11-22
CPU-Timer Validity . 11-22
Clock-Comparator Validity . 11-22
Machine-Check Extended Interruption
Information . 11-22
Register-Save Areas . 11-22
External-Damage Code . 11-23
Chapter 12. Operator Facilities

Manual Operation
Basic Operator Facilities
- Address-Compare Controls
- Alter-and-Display Controls
- Architectural-Mode Indicator
- Architectural-Mode-Selection Controls
- Check-Stop Indicator
- IML Controls
- Interrupt Key
- Load Indicator
- Load-Clear Key
- Load-Clear-List-Directed Key
- Load-Normal Key
- Load-with-Dump Key
- Load-Unit-Address Controls
- Manual Indicator
- Power Controls
- Rate Control
- Restart Key
- Start Key
- Stop Key
- Store-Status Key
- System-Reset-Clear Key
- System-Reset-Normal Key
- Test Indicator
- TOD-Clock Control
- Wait Indicator
- Multiprocessing Configurations

Chapter 13. I/O Overview

Input/Output (I/O)
The Channel Subsystem
- Subchannels
Attachment of Input/Output Devices
- Channel Paths
- Control Units
- I/O Devices

I/O Addressing
- Channel-Path Identifier
- Subchannel Number
- Device Number
- Device Identifier

Execution of I/O Operations
- Start-Function Initiation
- Path Management
- Channel-Program Execution
- Conclusion of I/O Operations
- I/O Interruptions

Chapter 14. I/O Instructions

I/O-Instruction Formats
I/O-Instruction Execution
- Serialization
- Operand Access
- Condition Code
- Program Exceptions

Instructions
- CANCEL SUBCHANNEL
- CLEAR SUBCHANNEL
- HALT SUBCHANNEL
- MODIFY SUBCHANNEL
- RESET CHANNEL PATH
- RESUME SUBCHANNEL
- SET ADDRESS LIMIT
- SET CHANNEL MONITOR
- START SUBCHANNEL
- STORE CHANNEL PATH STATUS
- STORE CHANNEL REPORT WORD
- STORE SUBCHANNEL
- TEST PENDING INTERRUPTION
- TEST SUBCHANNEL

Chapter 15. Basic I/O Functions

Control of Basic I/O Functions
- Subchannel-Information Block
- Path-Management-Control Word
- Subchannel-Status Word
- Model-Dependent Area/Measurement Block Address
- Summary of Modifiable Fields

Channel-Path Allegiance
- Working Allegiance
- Active Allegiance
- Dedicated Allegiance

Channel-Path Availability
- Control-Unit Type

Clear Function
- Clear-Function Path Management
- Clear-Function Subchannel Modification
- Clear-Function Signaling and Completion
Chapter 17. Device-Connect-Time-Measurement

Mode .................................. 17-10
Device-Connect-Time-Measurement Enable ................. 17-11
Extended Measurement Word .......................... 17-11
Extended-Measurement-Word Enable ....................... 17-11

Signals and Resets ................................ 17-12
Signals ..................................... 17-12
Halt Signal .................................. 17-12
Clear Signal .................................. 17-12
Reset Signal .................................. 17-13
Resets ...................................... 17-13
Channel-Path Reset ................................ 17-13
I/O-System Reset ................................ 17-13

Externally Initiated Functions ......................... 17-17
CCW-type IPL ................................ 17-18
List-Directed IPL ................................ 17-20

Reconfiguration of the I/O System .................. 17-22
Status Verification ................................ 17-23
Address-Limit Checking ............................ 17-23
Configuration Alert ............................... 17-24
Incorrect-Length-Indication Suppression .............. 17-24
Concurrent Sense ................................ 17-24
Channel-Subsystem Recovery ......................... 17-24
Channel Report ................................ 17-25
Channel-Report Word ................................ 17-26
Channel-Subchannel-Logout Facility ................. 17-28
Channel-Subsystem-I/O-Priority Facility ............. 17-29
Number of Channel-Subsystem-Priority Levels ........ 17-29

Chapter 18. Hexadecimal-Floating-Point

Instructions .................................. 18-1
HFP Arithmetic ................................ 18-1
HFP Number Representation .......................... 18-1
Normalization ................................ 18-3
HFP Data Format ................................ 18-3
Instructions .................................. 18-4
ADD NORMALIZED ............................. 18-8
ADD UNNORMALIZED ............................ 18-10
COMPARE .................................. 18-10
CONVERT FROM FIXED ......................... 18-11
CONVERT TO FIXED ........................... 18-11
DIVIDE ................................ 18-12
HALVE .................................. 18-13
LOAD AND TEST ................................ 18-14
LOAD COMPLEMENT ............................ 18-14
LOAD FP INTEGER ............................ 18-15
LOAD LENGTHENED ............................ 18-15
LOAD NEGATIVE ................................ 18-16
LOAD POSITIVE ................................ 18-16
LOAD ROUNDED ................................ 18-17
MULTIPLY .................................. 18-18
MULTIPLY AND ADD ............................ 18-19
MULTIPLY AND SUBTRACT ......................... 18-20
SQUARE ROOT ................................ 18-21
SUBTRACT NORMALIZED ......................... 18-22
SUBTRACT UNNORMALIZED ......................... 18-22

Chapter 19. Binary-Floating-Point

Instructions .................................. 19-1
Binary-Floating-Point Facility ....................... 19-1
Floating-Point-Control (FPC) Register ................. 19-2
IEEE Masks and Flags ................................ 19-3
FPC DXC Byte ................................ 19-3
Operations on the FPC Register ...................... 19-3

BFP Arithmetic ................................ 19-4
BFP Data Formats ................................ 19-4
BFP Short Format ................................ 19-4
BFP Long Format ................................ 19-4
BFP Extended Format ................................ 19-4
Biased Exponent ................................ 19-4
Significand .................................. 19-4
Values of Nonzero Numbers ......................... 19-4
Classes of BFP Data ................................ 19-5
Zeros ...................................... 19-6
Denormalized Numbers ................................ 19-6
Normal Numbers ................................ 19-6
Infinities .................................. 19-6
Signaling and Quiet NaNs ........................... 19-6
BFP-Format Conversion ............................ 19-7
BFP Rounding ................................ 19-7
Rounding Mode ................................ 19-7
Normalization and Denormalization ................. 19-8
BFP Comparison ................................ 19-8
Condition Codes for BFP Instructions ............... 19-9
Remainder .................................. 19-9
IEEE Exception Conditions ......................... 19-10
IEEE Invalid Operation .......................... 19-10
IEEE Division-By-Zero ............................ 19-11
IEEE Overflow ................................ 19-11
IEEE Underflow ................................ 19-12
IEEE Inexact ................................ 19-12

Result Figures ................................ 19-13
Data-Exception Codes (DXC) and Abbreviations .... 19-14

Instructions .................................. 19-14
ADD ................................ 19-18
COMPARE .................................. 19-23
COMPARE AND SIGNAL ......................... 19-24
CONVERT FROM FIXED ......................... 19-26
CONVERT TO FIXED ........................... 19-26
DIVIDE ................................ 19-29
DIVIDE TO INTEGER ............................ 19-29
EXTRACT FPC ................................ 19-33
LOAD AND TEST ................................ 19-34
LOAD COMPLEMENT ............................ 19-34
<table>
<thead>
<tr>
<th>Instruction/Example</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD HALFWORD (AH)</td>
<td>A-7</td>
</tr>
<tr>
<td>AND (N, NC, NI, NR)</td>
<td>A-8</td>
</tr>
<tr>
<td>rine (NI Example)</td>
<td>A-8</td>
</tr>
<tr>
<td>Linkage Instructions (BAL, BALR, BASR, BASR, BASSM, BSM)</td>
<td>A-8</td>
</tr>
<tr>
<td>Other BALR and BASR Examples</td>
<td>A-9</td>
</tr>
<tr>
<td>BRANCH AND STACK (BAKR)</td>
<td>A-10</td>
</tr>
<tr>
<td>BAKR Example 1</td>
<td>A-10</td>
</tr>
<tr>
<td>BAKR Example 2</td>
<td>A-11</td>
</tr>
<tr>
<td>BAKR Example 3</td>
<td>A-11</td>
</tr>
<tr>
<td>BRANCH ON CONDITION (BC, BCR)</td>
<td>A-11</td>
</tr>
<tr>
<td>BRANCH ON COUNT (BTC, BCTR)</td>
<td>A-12</td>
</tr>
<tr>
<td>BRANCH ON INDEX HIGH (BXH)</td>
<td>A-12</td>
</tr>
<tr>
<td>BXH Example 1</td>
<td>A-12</td>
</tr>
<tr>
<td>BXH Example 2</td>
<td>A-13</td>
</tr>
<tr>
<td>BRANCH ON INDEX LOW OR EQUAL (BXLE)</td>
<td>A-13</td>
</tr>
<tr>
<td>BXLE Example 1</td>
<td>A-13</td>
</tr>
<tr>
<td>BXLE Example 2</td>
<td>A-14</td>
</tr>
<tr>
<td>COMPARE AND FORM CODEWORD (CFC)</td>
<td>A-14</td>
</tr>
<tr>
<td>COMPARE HALFWORD (CH)</td>
<td>A-14</td>
</tr>
<tr>
<td>COMPARE LOGICAL (CL, CLC, CLI, CLR)</td>
<td>A-14</td>
</tr>
<tr>
<td>COMPARE LOGICAL CHARACTERS UNDER MASK (CLM)</td>
<td>A-15</td>
</tr>
<tr>
<td>COMPARE LOGICAL LONG (CLCL)</td>
<td>A-16</td>
</tr>
<tr>
<td>COMPARE LOGICAL STRING (CLST)</td>
<td>A-17</td>
</tr>
<tr>
<td>CONVERT TO BINARY (CVB)</td>
<td>A-18</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (CVD)</td>
<td>A-18</td>
</tr>
<tr>
<td>DIVIDE (D, DR)</td>
<td>A-19</td>
</tr>
<tr>
<td>EXCLUSIVE OR (X, XC, XI, XR)</td>
<td>A-19</td>
</tr>
<tr>
<td>XC Example</td>
<td>A-19</td>
</tr>
<tr>
<td>XI Example</td>
<td>A-20</td>
</tr>
<tr>
<td>EXECUTE (EX)</td>
<td>A-21</td>
</tr>
<tr>
<td>INSERT CHARACTERS UNDER MASK (ICM)</td>
<td>A-21</td>
</tr>
<tr>
<td>LOAD (L, LR)</td>
<td>A-22</td>
</tr>
<tr>
<td>LOAD ADDRESS (LA)</td>
<td>A-22</td>
</tr>
<tr>
<td>LOAD HALFWORD (LH)</td>
<td>A-23</td>
</tr>
<tr>
<td>MOVE (MVC, MVI)</td>
<td>A-23</td>
</tr>
<tr>
<td>MVC Example</td>
<td>A-23</td>
</tr>
<tr>
<td>MVI Example</td>
<td>A-24</td>
</tr>
<tr>
<td>MOVE INVERSE (MVCIN)</td>
<td>A-24</td>
</tr>
<tr>
<td>MOVE LONG (MVCL)</td>
<td>A-25</td>
</tr>
<tr>
<td>MOVE NUMERICs (MVN)</td>
<td>A-25</td>
</tr>
<tr>
<td>MOVE STRING (MVST)</td>
<td>A-26</td>
</tr>
<tr>
<td>MOVE WITH OFFSET (MVO)</td>
<td>A-26</td>
</tr>
<tr>
<td>MOVE ZONES (MVZ)</td>
<td>A-27</td>
</tr>
<tr>
<td>MULTIPLY (M, MR)</td>
<td>A-27</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD (MH)</td>
<td>A-27</td>
</tr>
<tr>
<td>OR (O, OC, OI, OR)</td>
<td>A-28</td>
</tr>
<tr>
<td>OI Example</td>
<td>A-28</td>
</tr>
<tr>
<td>PACK (PACK)</td>
<td>A-28</td>
</tr>
<tr>
<td>SEARCH STRING (SRST)</td>
<td>A-29</td>
</tr>
<tr>
<td>SRST Example 1</td>
<td>A-29</td>
</tr>
<tr>
<td>SRST Example 2</td>
<td>A-29</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE (SLDA)</td>
<td>A-29</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE (SLA)</td>
<td>A-30</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (STCM)</td>
<td>A-30</td>
</tr>
<tr>
<td>STORE MULTIPLE (STM)</td>
<td>A-30</td>
</tr>
<tr>
<td>TEST UNDER MASK (TM)</td>
<td>A-31</td>
</tr>
<tr>
<td>TRANSLATE (TR)</td>
<td>A-31</td>
</tr>
<tr>
<td>TRANSLATE AND TEST (TRT)</td>
<td>A-32</td>
</tr>
<tr>
<td>UNPACK (UNPK)</td>
<td>A-33</td>
</tr>
<tr>
<td>UPDATE TREE (UPT)</td>
<td>A-34</td>
</tr>
<tr>
<td>Decimal Instructions</td>
<td>A-34</td>
</tr>
<tr>
<td>ADD DECIMAL (AP)</td>
<td>A-34</td>
</tr>
<tr>
<td>COMPARE DECIMAL (CP)</td>
<td>A-34</td>
</tr>
<tr>
<td>DIVIDE DECIMAL (DP)</td>
<td>A-34</td>
</tr>
<tr>
<td>EDIT (ED)</td>
<td>A-35</td>
</tr>
<tr>
<td>EDIT AND MARK (EDMK)</td>
<td>A-36</td>
</tr>
<tr>
<td>MULTIPLY DECIMAL (MP)</td>
<td>A-36</td>
</tr>
<tr>
<td>SHIFT AND ROUND DECIMAL (SRP)</td>
<td>A-37</td>
</tr>
</tbody>
</table>
Notices

References in this publication to IBM* products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY, 10504-1785 USA.

Trademarks

The following terms, denoted by an asterisk (*) at the first or most prominent occurrence in this publication, are trademarks of the International Business Machines Corporation in the United States or other countries:

- AIX/ESA
- BookMaster
- CICS
- DB2
- Enterprise Systems Architecture/370
- Enterprise Systems Architecture/390
- Enterprise Systems Connection Architecture
- ESA/370
- ESA/390
- ESCON
- FICON
- IBM
- IBMLink
- MVS/ESA
- OS/390
- Processor Resource/Systems Manager
- PR/SM
- Sysplex Timer
- System/370
- VM/ESA
- z/Architecture
- z/OS
Preface

This publication provides, for reference purposes, a detailed z/Architecture* description.

The publication applies only to systems operating as defined by z/Architecture. For systems operating in accordance with the Enterprise Systems Architecture/390* (ESA/390*) definition, the IBM ESA/390 Principles of Operation, SA22-7201, should be consulted.

The publication describes each function at the level of detail needed to prepare an assembler-language program that relies on that function. It does not, however, describe the notation and conventions that must be employed in preparing such a program, for which the user must instead refer to the appropriate assembler-language publication.

The information in this publication is provided principally for use by assembler-language programmers, although anyone concerned with the functional details of z/Architecture will find it useful.

This publication is written as a reference and should not be considered an introduction or a textbook. It assumes the user has a basic knowledge of data-processing systems.

All facilities discussed in this publication are not necessarily available on every model. Furthermore, in some instances the definitions have been structured to allow for some degree of extendibility, and therefore certain capabilities may be described or implied that are not offered on any model. Examples of such capabilities are the use of a 16-bit field in the subsystem-identification word to identify the subchannel number, the size of the CPU address, and the number of CPUs sharing main storage. The allowance for this type of extendibility should not be construed as implying any intention by IBM to provide such capabilities. For information about the characteristics and availability of facilities on a specific model, see the functional characteristics publication for that model.

Largely because this publication is arranged for reference, certain words and phrases appear, of necessity, earlier in the publication than the principal discussions explaining them. The reader who encounters a problem because of this arrangement should refer to the index, which indicates the location of the key description.

The information presented in this publication is grouped in 19 chapters and several appendixes:

Chapter 1, Introduction, highlights the major facilities of z/Architecture.

Chapter 2, Organization, describes the major groupings within the system — main storage, expanded storage, the central processing unit (CPU), the external time reference (ETR), and input/output — with some attention given to the composition and characteristics of those groupings.

Chapter 3, Storage, explains the information formats, the addressing of storage, and the facilities for storage protection. It also deals with dynamic address translation (DAT), which, coupled with special programming support, makes the use of a virtual storage possible.

Chapter 4, Control, describes the facilities for the switching of system status, for special externally initiated operations, for debugging, and for timing. It deals specifically with CPU states, control modes, the program-status word (PSW), control registers, tracing, program-event recording, timing facilities, resets, store status, and initial program loading.

Chapter 5, Program Execution, explains the role of instructions in program execution, looks in detail at instruction formats, and describes briefly the use of the program-status word (PSW), of branching, and of interruptions. It contains the principal description of the advanced address-space facilities that were introduced in ESA/370*. It also details the aspects of program execution on one
CPU as observed by other CPUs and by channel programs.

Chapter 6, Interruptions, details the mechanism that permits the CPU to change its state as a result of conditions external to the system, within the system, or within the CPU itself. Six classes of interruptions are identified and described: machine-check interruptions, program interruptions, supervisor-call interruptions, external interruptions, input/output interruptions, and restart interruptions.

Chapter 7, General Instructions, contains detailed descriptions of logical and binary-integer data formats and of all unprivileged instructions except the decimal and floating-point instructions.

Chapter 8, Decimal Instructions, describes in detail decimal data formats and the decimal instructions.

Chapter 9, Floating-Point Overview and Support Instructions, includes an introduction to the floating-point operations, detailed descriptions of those instructions common to both hexadecimal-floating-point and binary-floating-point operations, and summaries of all floating-point instructions.

Chapter 10, Control Instructions, contains detailed descriptions of all of the semiprivileged and privileged instructions except for the I/O instructions.

Chapter 11, Machine-Check Handling, describes the mechanisms for detecting, correcting, and reporting machine malfunctions.

Chapter 12, Operator Facilities, describes the basic manual functions and controls available for operating and controlling the system.

Chapters 13-17 of this publication provide a detailed definition of the functions performed by the channel subsystem and the logical interface between the CPU and the channel subsystem.

Chapter 13, I/O Overview, provides a brief description of the basic components and operation of the channel subsystem.

Chapter 14, I/O Instructions, contains the description of the I/O instructions.

Chapter 15, Basic I/O Functions, describes the basic I/O functions performed by the channel subsystem, including the initiation, control, and conclusion of I/O operations.

Chapter 16, I/O Interruptions, covers I/O interruptions and interruption conditions.

Chapter 17, I/O Support Functions, describes such functions as channel-subsystem usage monitoring, resets, initial-program loading, reconfiguration, and channel-subsystem recovery.

Chapter 18, Hexadecimal-Floating-Point Instructions, contains detailed descriptions of the hexadecimal-floating-point (HFP) data formats and the HFP instructions.

Chapter 19, Binary-Floating-Point Instructions, contains detailed descriptions of the binary-floating-point (BFP) data formats and the BFP instructions.

The Appendixes include:
- Information about number representation
- Instruction-use examples
- Lists of the instructions arranged in several sequences
- A summary of the condition-code settings
- A table of the powers of 2
- Tabular information helpful in dealing with hexadecimal numbers
- A table of EBCDIC and other codes.

Size and Number Notation

In this publication, the letters K, M, G, T, P, and E denote the multipliers $2^{10}$, $2^{20}$, $2^{30}$, $2^{40}$, $2^{50}$, and $2^{60}$, respectively. Although the letters are borrowed from the decimal system and stand for kilo ($10^3$), mega ($10^6$), giga ($10^9$), tera ($10^{12}$), peta ($10^{15}$), and exa ($10^{18}$), they do not have the decimal meaning but instead represent the power of 2 closest to the corresponding power of 10. Their meaning in this publication is as follows:
The following are some examples of the use of K, M, G, T, and E:

\[
\begin{align*}
2,048 & \text{ is expressed as } 2K. \\
4,096 & \text{ is expressed as } 4K. \\
65,536 & \text{ is expressed as } 64K \text{ (not } 65K). \\
2^{24} & \text{ is expressed as } 16M. \\
2^{31} & \text{ is expressed as } 2G. \\
2^{42} & \text{ is expressed as } 4T. \\
2^{64} & \text{ is expressed as } 16E.
\end{align*}
\]

When the words “thousand” and “million” are used, no special power-of-2 meaning is assigned to them.

All numbers in this publication are in decimal unless they are explicitly noted as being in binary or hexadecimal (hex).

## Bytes, Characters, and Codes

Although the System/360 architecture was originally designed to support the Extended Binary-Coded-Decimal Interchange Code (EBCDIC), the instructions and data formats of the architecture are for the most part independent of the external code which is to be processed by the machine. For most instructions, all 256 possible combinations of bit patterns for a particular byte can be processed, independent of the character which the bit pattern is intended to represent. For instructions which use the zoned format, and for those few instructions which are dependent on a particular external code, the instruction TRANSLATE may be used to convert data from one code to another code. Thus, a machine operating in accordance with z/Architecture can process EBCDIC, ASCII, or any other code which can be represented in eight or fewer bits per character.

In this publication, unless otherwise specified, the value given for a byte is the value obtained by considering the bits of the byte to represent a binary code. Thus, when a byte is said to contain a zero, the value 00000000 binary, or 00 hex, is meant, and not the value for an EBCDIC character “0,” which would be F0 hex.

## Other Publications

The parallel-I/O interface is described in the publication IBM System/360 and System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers’ Information, GA22-6974.

The parallel-I/O channel-to-channel adapter is described in the publication IBM Enterprise Systems Architecture/390 Channel-to-Channel Adapter for the System/360 and System/370 I/O Interface, SA22-7091.

The Enterprise Systems Connection Architecture* (ESCON*) I/O interface, referred to in this publication along with the FICON I/O interface as the serial-I/O interface, is described in the publication IBM Enterprise Systems Architecture/390 ESCON I/O Interface, SA22-7202.

The FICON I/O interface is described in the ANSI standards document Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2).

The channel-to-channel adapter for the serial-I/O interface is described in the publication IBM Enterprise Systems Architecture/390 ESCON Channel-to-Channel-Adapter, SA22-7203.

The commands, status, and sense data that are common to all I/O devices that comply with z/Architecture are described in the publication IBM Enterprise Systems Architecture/390 Common I/O-Device Commands and Self Description, SA22-7204.

The compression facility is described in the publication IBM Enterprise Systems Architecture/390 Data Compression, SA22-7208. The

---

*Enterprise Systems Connection Architecture and ESCON are trademarks of the International Business Machines Corporation.
ANSI is a registered trademark of the American National Standards Institute.
z/Architecture form of the COMPRESSION CALL instruction is described in this publication.

The interpretive-execution facility is described in the publication IBM 370-XA Interpretive Execution, SA22-7095.

Summary of Changes in Fourth Edition

The current, fourth edition of this publication differs from the previous edition principally by containing the definitions of the extended-translation facility 3 and the ASN-and-LX-reuse facility. The fourth edition contains minor clarifications and corrections and also the following significant changes relative to the previous edition:

- In Chapter 3, "Storage":
  - Changes introduced by the ASN-and-LX-reuse facility are added, including changes to the "Address Spaces" and "ASN Translation" sections.
  - A programming note is added to the "Dynamic Address Translation" section, describing implications in using common segments.
  - The ATMID and AI fields are corrected in the "Assigned Storage Locations" figure.

- In Chapter 4, "Control":
  - Changes introduced by the ASN-and-LX-reuse facility are added, including changes to the "Trace" section.
  - The list-directed IPL function is added to the "Initial Program Load" section.
  - In the "Trace" section, serialization requirements for instructions that implicitly store into the trace table or linkage stack are relaxed.

- In Chapter 5, "Program Execution":
  - A new RI and SS instruction format are included.
  - Changes introduced by the ASN-and-LX-reuse facility are added, including changes to the "Authorization Mechanisms," "PC-Number Translation," and "Linkage-Stack Operations" sections.
  - The instructions of the extended-translation facility 3 (except TRANSLATE AND TEST REVERSED) are added to the sections "Condition-Code Alternative to Interruptibility" and "Multiple-Access References."
  - Serialization requirements for instructions that implicitly store into the trace table or linkage stack are relaxed.

- In Chapter 6, "Interruptions":
  - Changes introduced by the ASN-and-LX-reuse facility include the new LFX translation, LSX translation, LSTE sequence, and ASTE instance exceptions.
  - An additional condition for TRAP is added to the list of instructions that can cause a special-operation exception to be recognized.
  - SEARCH STRING UNICODE, a part of the extended-translation facility 3, is added to the list of instructions that can cause a specification exception to be recognized.

- In Chapter 7, "General Instructions":
  - Six new instructions provided by the extended-translation facility 3 are added. The instructions CONVERT UNICODE TO UTF-8 (CUUTF) and CONVERT UTF-8 TO UNICODE (CUTFU) are renamed to CONVERT UTF-16 TO UTF-8 (CU21) and CONVERT UTF-8 TO UTF-16 (CU12), respectively. The old mnemonics continue to be recognized.
  - The instruction-format illustrations for STAMY and STMY are corrected.

- In Chapter 10, "Control Instructions":
  - Changes introduced by the ASN-and-LX-reuse facility include the following:
    - Four new instructions provided by the facility are added.
    - The definitions of BRANCH AND STACK, BRANCH IN SUBSPACE GROUP, EXTRACT PRIMARY ASN, EXTRACT SECONDARY ASN, LOAD ADDRESS SECONDARY ASN, LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, and SET SECONDARY ASN are updated to account for the facility.
– New facility bits for the ASN-and-LX-reuse facility and the extended-translation facility 3 are added to STORE FACILITY LIST.

– New fields are added to the system-information block (SYSIB) returned by STORE SYSTEM INFORMATION.

– Serialization requirements for instructions that implicitly store into the trace table or linkage stack are relaxed, including PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER (WITH INSTANCE), SET SECONDARY ASN (WITH INSTANCE), and TRACE.

– Corrections and clarifications are made to TRAP.

In Chapter 12, “Operator Facilities”: Changes introduced by the list-directed IPL function are added, including the load-clear-list-directed key and the load-with-dump key.

In Chapter 14, “I/O Instructions,” Clarifications are added to programming note 1 of the CANCEL SUBCHANNEL instruction description.

In Chapter 16, “I/O Interruptions”:

– Bit 1 of the extended-report word (ERW) is defined as the request-logging-only (L) bit.

– Bit 2 of the ERW is defined as the extended-subchannel-logout-pending (E) bit.

– Words 2-3 of the format-0 extended-status word (ESW) are defined as the failing-storage address when the failing-storage-address-validity (F) bit, bit 6 of the ERW, is one and as the extended-subchannel-logout descriptor (ESLD) when the extended-subchannel-logout-pending (E) bit, bit 2 of the ERW, is one. The E-bit is always zero when the F-bit is one, and the F-bit is always zero when the E-bit is one.

In Appendix C, “Condition-Code Settings”: The instructions of the extended-translation facility 3 are added.

Summary of Changes in Third Edition

The third edition of this publication differs from the previous edition principally by containing the definitions of the DAT-enhancement, HFP-multiply-add/subtract, and long-displacement facilities and the message-security assist. The third edition contains minor clarifications and corrections and also the following significant changes relative to the previous edition:

– In Chapter 3, “Storage”:
  – Clarifications are added to the description of dynamic-address-translation process.

– The primary address-space-control element (ASCE) in control register 1 is an attaching ASCE even when the CPU is in the home-space mode, and the home ASCE in control register 13 is an attaching ASCE even when the CPU is in the secondary-space mode.

– The illustration of the PER code in Figure 3-12 on page 3-65 is corrected.

– The relationships between ETR time (TOD-clock time), UTC, and International Atomic Time are described in a programming note on page 4-42.

– Code 0 of the SIGNAL PROCESSOR set-architecture order, and also a CPU reset due to activation of the load normal key, are changed to save the current z/Architecture PSW when switching to the ESA/390 architectural mode. Also, code 2 of the order is added, and this restores, for CPUs other than the one executing SIGNAL PROCESSOR, the saved PSW when switching to the z/Architecture architectural mode, provided that the saved PSW has not been set to all zeros by certain resets.

– In Chapter 5, “Program Execution”:
  – The I format, the RI format with a M₁ operand, and the SS format with the I₃ operand are added.
- The RSY, RXY, and SIY instruction formats are added, and the RSE format is deleted. (All instructions that were of format RSE are now referred to as being of format RSY.)

- The formation of an operand address using the 20-bit signed displacement of instructions of formats RSY, RXY, and SIY is described.

- The results when a PER instruction-fetching event occurs along with certain exceptions or exception conditions are clarified. See "Indication of PER Events Concurrently with Other Interruption Conditions" on page 4-35.

- The fetch of the address-space-control element from the ASN-second-table entry during access-register translation is doubleword concurrent as observed by other CPUs.

- The change bit is not necessarily set to one currently with the related storage reference, as observed by other CPUs; it may be set to one before or after the reference, within certain limits. See "Storage-Key Accesses" on page 5-95 for a detailed description of when the change bit is set.

- The five instructions of the message-security assist are added to the list of instructions having multiple-access references.

- In Chapter 6, "Interruptions," the list of conditions causing a specification exception to be recognized is extended to include those caused by the message-security assist instructions.

- In Chapter 7, "General Instructions":
  - Thirty-nine instructions provided by the long-displacement facility are added. With the exception of the new LOAD BYTE instruction, the instructions added by the long-displacement facility have names and functions that are the same as existing instructions (but the mnemonics and opcodes are new). The new instructions are of formats RSY, RXY, and SIY and have a 20-bit signed displacement instead of a 12-bit unsigned displacement.

- All previously existing format-RSE and format-RXE instructions are changed to be of formats RSY and RXY, respectively, by use of a previously unused byte in the instructions. These changes are not marked by a bar in the margin.

- Five instructions provided by the message-security assist are added.

- The instruction format of SUPERVISOR CALL is changed to I.

- In Chapter 9, “Floating-Point Overview and Support Instructions,” four instructions provided by the long-displacement facility are added. These are the LOAD (long and short) and STORE (long and short) instructions.

- In Chapter 10, “Control Instructions”:
  - The COMPARE AND SWAP AND PURGE (CSPG) and INVALIDATE DAT TABLE ENTRY instructions provided by the DAT-enhancement facility are added. CSPG operates on a doubleword operand in storage.

- The definition of LOAD ADDRESS SPACE PARAMETERS is clarified.

- The LOAD REAL ADDRESS (LRAY) instruction provided by the long-displacement facility is added.

- All previously existing format-RSE instructions are changed to be of format RSY by use of a previously unused byte in the instructions. These changes are not marked by a bar in the margin.

- The description of the bits set by STORE FACILITY LIST is clarified, and new bits are assigned.

- In Chapter 14, “I/O Instructions”:
  - The definition of MODIFY SUBCHANNEL is modified.

- The definition of SET CHANNEL MONITOR is modified.

- In Chapter 15, “Basic I/O Functions,” the following changes are made to the subchannel-information-block (SCHIB):
  - Bit 29 of word 6 of the path-management-control word (PMCW) is defined as the measurement-block-format control.
Bit 30 of word 6 of the PMCW is defined as the extended-measurement-word-mode enablement bit.

The definition of words 10-11 (words 0-1 of the model-dependent area) are changed to contain a measurement-block address, when the extended-I/O-measurement-block facility is installed.

In Chapter 16, “I/O Interruptions,” the interruption-response block (IRB) is extended to include the extended-measurement word.

In Chapter 17, “I/O Support Functions”:
- The requirement that the measurement block be updated when secondary status is accepted is clarified.
- The extended-measurement-block facility is added.
- The extended-measurement-word facility is added.

In Chapter 18, “Hexadecimal-Floating-Point Instructions,” the MULTIPLY AND ADD (four instructions) and MULTIPLY AND SUBTRACT (four instructions) instructions provided by the HFP-multiply-add/subtract facility are added.

The above changes may affect other chapters besides the ones listed. All technical changes to the text or to an illustration are indicated by a vertical line to the left of the change.

Summary of Changes in Second Edition

The second edition of this publication differs from the previous edition mainly by containing clarifications and corrections. The significant changes are as follows:

In Chapter 1, “Introduction”:
- Summaries of DIVIDE LOGICAL and MULTIPLY LOGICAL, TEST ADDRESSING MODE, the set-architecture order of SIGNAL PROCESSOR, and STORE FACILITY LIST are added or improved.
- An extensive summary of the input/output enhancements placed in z/Architecture is added.

In Chapter 3, “Storage”:
- Definitions of absolute locations 0-23 are deleted since they pertain only to an ESA/390 initial program load.
- The definition of real locations 200-203, stored in by STORE FACILITY LIST, is corrected to state that bit 16 indicates the extended-translation facility 2.
- In Chapter 4, “Control,” a description of unassigned fields in the PSW is corrected to state that bit 4 is unassigned and bit 31 is assigned.
- In Chapter 5, “Program Execution,” the RSL format and an RIL format with an M1 field are added.
- In Chapter 7, “General Instructions”:
  - The definition of BRANCH AND SET MODE is corrected to state that bit 63 of the R1 general register remains unchanged in the 24-bit or 31-bit addressing mode; the bit is not set to zero.
  - The definitions of PACK ASCII, PACK UNICODE, UNPACK ASCII, and UNPACK UNICODE are clarified.
  - It is clarified that the following instructions perform multiple-access references to their storage operands:
    - CHECKSUM
    - COMPARE AND FORM CODEWORD
    - CONVERT UNICODE TO UTF-8
    - CONVERT UTF-8 TO UNICODE
  - It is clarified that the following instructions do not necessarily process their storage operands left to right as observed by other CPUs: MOVE LONG, MOVE LONG EXTENDED, and MOVE LONG UNICODE. Special padding characters of MOVE LONG and MOVE LONG EXTENDED specify whether left-to-right processing should be performed, as observed by other CPUs, and whether the data being moved should or should not be placed in the cache for availability for subsequent processing.

In Chapter 10, “Control Instructions,” it is clarified that the following instructions perform multiple-access references to their storage operands:
- LOAD ADDRESS SPACE PARAMETERS
Chapters 13-17 contain many clarifying changes, all indicated by a vertical line in the margin, in addition to the significant changes listed below.

- In Chapter 13, “I/O Overview,” statements about the suspend flag in a CCW are clarified to describe the flag being specified as a one and being valid because of a one value of the suspend control in the associated ORB.
- In Chapter 14, “I/O Instructions,” the results of MODIFY SUBCHANNEL when the device-number-valid bit at the designated subchannel is zero are corrected.
- In Chapter 15, “Basic I/O Functions”:
  - It is clarified that unlimited prefetching of data and IDAWs associated with the current and prefetched CCWs is allowed independent of the value of the prefetch control in the associated ORB.
  - A specified control-unit-priority number is ignored if the channel-subsystem-I/O-priority facility is not operational due to an operator action.
- In Chapter 16, “I/O Interruptions,” the form of the address stored in the failing-storage-address field is described in terms of the format-2-IDAW control instead of an addressing mode.
- In Chapter 17, “I/O Support Functions”:
  - The introduction to the channel-subsystem monitoring facilities is clarified.
  - References to the measurement block by the measurement-block-update facility are single-access references and appear to be word concurrent as observed by CPUs. They do not appear to be block concurrent.
  - The description of the channel-subsystem-I/O-priority facility is corrected by including mention of control-unit priority for fibre-channel-attached control units.

The above changes may affect other chapters besides the ones listed.
Chapter 1. Introduction

This publication provides, for reference purposes, a detailed description of z/Architecture.

The architecture of a system defines its attributes as seen by the programmer, that is, the conceptual structure and functional behavior of the machine, as distinct from the organization of the data flow, the logical design, the physical design, and the performance of any particular implementation. Several dissimilar machine implementations may conform to a single architecture. When the execution of a set of programs on different machine implementations produces the results that are defined by a single architecture, the implementations are considered to be compatible for those programs.

Highlights of Original z/Architecture

z/Architecture is the next step in the evolution from the System/360 to the System/370*, System/370 extended architecture (370-XA), Enterprise Systems Architecture/370* (ESA/370), and Enterprise Systems Architecture/390 (ESA/390). z/Architecture includes all of the facilities of ESA/390 except for the asynchronous-pageout, asynchronous-data-mover, program-call-fast, and vector facilities. z/Architecture also provides significant extensions, as follows:

- Sixty-four-bit general registers and control registers.
- A 64-bit addressing mode, in addition to the 24-bit and 31-bit addressing modes of ESA/390, which are carried forward to z/Architecture.

Both operand addresses and instruction addresses can be 64-bit addresses. The program-status word (PSW) is expanded to 16 bytes to contain the larger instruction address. The PSW also contains a newly assigned bit that specifies the 64-bit addressing mode.

- Up to three additional levels of dynamic-address-translation (DAT) tables, called region tables, for translating 64-bit virtual addresses.

A virtual address space may be specified either by a segment-table designation as in ESA/390 or by a region-table designation, and either of these types of designation is called an address-space-control element (ASCE). An ASCE may alternatively be a real-space designation that causes virtual addresses to be treated simply as real addresses without the use of DAT tables.

---

System/370 and Enterprise Systems Architecture/370 are trademarks of the International Business Machines Corporation.

© Copyright IBM Corp. 1990-2004

1-1
• An 8K-byte prefix area for containing larger old and new PSWs and register save areas.

• A SIGNAL PROCESSOR order for switching between the ESA/390 and z/Architecture architectural modes.

Initial program loading sets the ESA/390 architectural mode. The new SIGNAL PROCESSOR order then can be used to set the z/Architecture mode or to return from z/Architecture to ESA/390. This order causes all CPUs in the configuration always to be in the same architectural mode.

• Many new instructions, many of which operate on 64-bit binary integers

Some of the new instructions that do not operate on 64-bit binary integers have also been added to ESA/390.

All of the ESA/390 instructions, except for those of the four facilities named above, are included in z/Architecture.

The bit positions of the general registers and control registers of z/Architecture are numbered 0-63. An ESA/390 instruction that operates on bit positions 0-31 of a 32-bit register in ESA/390 operates instead on bit positions 32-63 of a 64-bit register in z/Architecture.

z/Architecture was announced in October, 2000. The remainder of this section summarizes the original contents of z/Architecture. Subsequent additions are described in "Additions to z/Architecture" on page 1-7.

General Instructions for 64-Bit Integers

The 32-bit-binary-integer instructions of ESA/390 have new analogs in z/Architecture that operate on 64-bit binary integers. There are two types of analogs:

• Analogs that use two 64-bit binary integers to produce a 64-bit binary integer. For example, the ESA/390 ADD instruction (A for a storagetoregister operation or AR for a register-toregister operation) has the analogs AG (adds 64 bits from storage to the contents of a 64-bit general register) and AGR (adds the contents of a 64-bit general register to the contents of another 64-bit general register). These analogs are distinguished by having “G” in their mnemonics.

• Analogs that use a 64-bit binary integer and a 32-bit binary integer to produce a 64-bit binary integer. The 32-bit integer is either sign-extended or extended on the left with zeros, depending on whether the operation is signed or unsigned, respectively. For example, the ESA/390 ADD (A or AR) instruction has the analogs AGF (adds 32 bits from storage to the contents of a 64-bit general register) and AGFR (adds the contents of bit positions 32-63 of a 64-bit general register to the contents of another 64-bit general register). These analogs are distinguished by having “GF” in their mnemonics.

Other New General Instructions

The other additional or significantly enhanced general instructions of z/Architecture are highlighted as follows:

• ADD LOGICAL WITH CARRY and SUBTRACT LOGICAL WITH BORROW operate on either 32-bit or 64-bit unsigned binary integers and include a carry or borrow, as represented by the leftmost bit of the two-bit condition code in the PSW, in the computation. This can improve the performance of operating on extended-precision integers (integers longer than 64 bits).

• AND IMMEDIATE and OR IMMEDIATE combine a two-byte immediate operand with any of the two bytes on two-byte boundaries in a 64-bit general register.

• BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE are enhanced so that they set bit 63 of the R1 general register to one if the current addressing mode is the 64-bit mode, and they set the 64-bit addressing mode if bit 63 of the R2 general register is one. This allows “pointer-directed” linkages between programs in different addressing modes, including any of the 24-bit, 31-bit, and 64-bit modes.

• BRANCH RELATIVE AND SAVE LONG and BRANCH RELATIVE ON CONDITION LONG are like the BRANCH RELATIVE AND SAVE and BRANCH RELATIVE ON CONDITION instructions of ESA/390 except that the new instructions use a 32-bit immediate field. This
increases the target range available through relative branching.

- **COMPARE AND FORM CODEWORD** is enhanced so that, in the 64-bit addressing mode, the comparison unit is six bytes instead of two and the resulting codeword is eight bytes instead of four. **UPDATE TREE** is enhanced so that, in the 64-bit addressing mode, a node is 16 bytes instead of eight and the codeword in a node is eight bytes instead of four. This improves the performance of sorting records having long keys.

- **DIVIDE LOGICAL** uses a 64-bit or 128-bit unsigned binary dividend and a 32-bit or 64-bit unsigned binary divisor, respectively, to produce a 32-bit or 64-bit quotient and remainder, respectively. **MULTIPLY LOGICAL** uses a 32-bit or 64-bit unsigned binary multiplicand and multiplier to produce a 64-bit or 128-bit product, respectively.

- **DIVIDE SINGLE** divides a 64-bit dividend by a 32-bit or 64-bit divisor and produces a 64-bit quotient and remainder. **MULTIPLY SINGLE** is enhanced so it can multiply a 64-bit multiplicand by a 32-bit or 64-bit multiplier and produce a 64-bit product.

- **EXTRACT PSW** extracts the entire current PSW to allow determination of the current machine state, for example, determination of whether the CPU is in the problem state or the supervisor state.

- **INSERT IMMEDIATE** inserts a two-byte immediate operand into a 64-bit general register on any of the two-byte boundaries in the register. **LOAD LOGICAL IMMEDIATE** does the same and also clears the remainder of the register.

- **LOAD ADDRESS RELATIVE LONG** forms an address relative to the current (unupdated) instruction address by means of a signed 32-bit immediate field.

- **LOAD LOGICAL THIRTY ONE BITS** places the rightmost 31 bits of either a general register or a word in storage, with 33 zeros appended on the left, in a general register.

- **LOAD MULTIPLE DISJOINT** loads the leftmost 32 bits of each register in a range of general registers from one area in storage and the rightmost 32 bits of each of those registers from another area in storage. This is for use in place of a **LOAD MULTIPLE HIGH instruction** and a 32-bit **LOAD MULTIPLE** instruction when one of the storage areas is addressed by one of the registers loaded.

- **LOAD MULTIPLE HIGH** and **STORE MULTIPLE HIGH** load or store the leftmost 32 bits of each register in a range of general registers, allowing augmentation of existing programs that load or store the rightmost 32 bits by means of **LOAD MULTIPLE** and **STORE MULTIPLE**. (Sixty-four-bit forms of **LOAD MULTIPLE** and **STORE MULTIPLE** also are provided.)

- **LOAD PAIR FROM QUADWORD** and **STORE PAIR TO QUADWORD** operate between an even-odd pair of 64-bit general registers and a quadword in storage (16 bytes aligned on a 16-byte boundary). These instructions provide quadword consistency (all bytes appear to be loaded or stored concurrently in a multiple-CPU system). (Only the 64-bit form of **COMPARE DOUBLE AND SWAP** also provides quadword consistency.)

- **LOAD REVERSED** and **STORE REVERSED** load or store a two-byte, four-byte, or eight-byte unit in storage with the left-to-right sequence of the bytes reversed. **LOAD REVERSED** also can move a four-byte or eight-byte unit between two general registers. These operations allow conversion between “little-endian” and “big-endian” formats.

- **PERFORM LOCKED OPERATION** is enhanced with two more sets of function codes, with each set providing six different operations. One of the additional sets provides operations on 64-bit operands in 64-bit general registers, and the other provides operations on 128-bit operands in a parameter list.

- **ROTATE LEFT SINGLE LOGICAL** obtains 32 bits or 64 bits from a general register, rotates them (the leftmost bit replaces the rightmost bit), and places the result in another general register (a nondestructive rotate).

- **SET ADDRESSING MODE** can set any of the 24-bit, 31-bit, and 64-bit addressing modes.

- **SHIFT LEFT SINGLE**, **SHIFT LEFT SINGLE LOGICAL**, **SHIFT RIGHT SINGLE**, and **SHIFT RIGHT SINGLE LOGICAL** are enhanced with 64-bit forms that obtain the source operand from one general register and place the result...
operand in another general register (a nonde-
structive shift).

- TEST ADDRESSING MODE sets the condi-
tion code to indicate whether bits 31 and 32 of
the current PSW specify the 24-bit, 31-bit, or
64-bit addressing mode.

- TEST UNDER MASK HIGH and TEST
UNDER MASK LOW, which are ESA/390
instructions, are given the alternative name
TEST UNDER MASK, and two additional
forms are added so that a two-byte immediate
operand can be used to test the bits of two
bytes located on any of the two-byte bounda-
ries in a 64-bit general register. (The
ESA/390 instruction TEST UNDER MASK,
which uses a one-byte immediate operand to
test a byte in storage, continues to be pro-
vided.)

Floating-Point Instructions

The z/Architecture floating-point instructions are
the same as in ESA/390 except that instructions
are added for converting between 64-bit signed
binary integers and either hexadecimal or binary
floating-point data. These new instructions have
“G” in their mnemonics.

Control Instructions

The new or enhanced control instructions of
z/Architecture are highlighted as follows:

- EXTRACT AND SET EXTENDED
AUTHORITY is a privileged instruction for
changing the extended authorization index in
a control register. This enables real-space
designations to be used more efficiently by
means of access lists.

- EXTRACT STACKED REGISTERS is
enhanced to extract optionally all 64 bits of
the contents of one or more saved general
registers.

- EXTRACT STACKED STATE is enhanced to
extract optionally the entire contents of the
saved PSW, including a 64-bit instruction
address.

- LOAD CONTROL and STORE CONTROL are
enhanced for operating optionally on 64-bit
control registers.

- LOAD PSW uses an eight-byte storage
operand as in ESA/390 and expands this
operand to a 16-byte z/Architecture PSW.

- LOAD PSW EXTENDED directly loads a
16-byte PSW.

- LOAD REAL ADDRESS in its ESA/390 form
and in the 24-bit or 31-bit addressing mode
operates as in ESA/390 if the translation is
successful and the obtained real address has
a value less than 2G bytes. LOAD REAL
ADDRESS in its ESA/390 form and in the
64-bit addressing mode, or in its enhanced
z/Architecture form in any addressing mode,
loads a 64-bit real address.

- LOAD USING REAL ADDRESS and STORE
USING REAL ADDRESS are enhanced to
have optionally 64-bit operands.

- SIGNAL PROCESSOR has a new order that
can be used to switch all CPUs in the config-
uration either from the ESA/390 architectural
mode to the z/Architecture architectural mode
or from z/Architecture to ESA/390. (A system
that is to operate using z/Architecture must
first be IPLed in the ESA/390 mode.)

- STORE FACILITY LIST is a privileged instruc-
tion that stores at real location 200 an indi-
cation of whether z/Architecture is installed
and of whether it is active. This instruction is
added also to ESA/390 and also stores an
indication of whether the new z/Architecture
instructions that have been added to ESA/390
are available. Real location 200 has previ-
ously contained all zeros in most systems and
normally can be examined by a problem-state
program whether or not STORE FACILITY
LIST is installed. The information stored at
real location 200 also indicates whether the
extended-translation facility 2 is installed.

- STORE REAL ADDRESS is like LOAD REAL
ADDRESS except that STORE REAL
ADDRESS stores the resulting address
instead of placing it in a register.

- TRACE is enhanced to record optionally the
contents of 64-bit general registers.
Trimodal Addressing

“Trimodal addressing” refers to the ability to switch between the 24-bit, 31-bit, and 64-bit addressing modes. This switching can be done by means of:

- The old instructions BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE. Both of these instructions set the 64-bit addressing mode if bit 63 of the R register is one. If bit 63 is zero, the instructions set the 24-bit or 31-bit addressing mode if bit 32 of the register is zero or one, respectively.
- The new instruction SET ADDRESSING MODE (SAM24, SAM31, and SAM64). The instruction sets the 24-bit, 31-bit, or 64-bit addressing mode as determined by the operation code.

Modal Instructions

Trimodal addressing affects the general instructions only in the manner in which logical storage addresses are handled, except as follows.

- The instructions BRANCH AND LINK, BRANCH AND SAVE, BRANCH AND SAVE AND SET MODE, BRANCH AND SET MODE, and BRANCH RELATIVE AND SAVE place information in bit positions 32-39 of general register R as in ESA/390 in the 24-bit or 31-bit addressing mode or place address bits in those bit positions in the 64-bit addressing mode. The new instruction BRANCH RELATIVE AND SAVE LONG does the same.
- The instructions BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE place a one in bit position 63 of general register R in the 64-bit addressing mode. In the 24-bit or 31-bit mode, BRANCH AND SAVE AND SET MODE sets bit 63 to zero, and BRANCH AND SET MODE leaves it unchanged.
- Certain instructions leave bits 0-31 of a general register unchanged in the 24-bit or 31-bit addressing mode but place or update address or length information in them in the 64-bit addressing mode. These are listed in a programming note on page 7-6 and are sometimes called modal instructions.

Effects on Bits 0-31 of a General Register

Bits 0-31 of general registers are changed by two types of instructions. The first type is a modal instruction (see the preceding section) when the instruction is executed in the 64-bit addressing mode. The second type is an instruction having, independent of the addressing mode, either a 64-bit result operand in a single general register or a 128-bit result operand in an even-odd general-register pair.

Most of the instructions of the second type are indicated by a “G,” either alone or in “GF,” in their mnemonics. The other instructions that change or may change bits 0-31 of a general register regardless of the current addressing mode are listed in a programming note on page 7-7. All of the instructions of the second type are sometimes referred to as “G-type” instructions.

If a program is not executed in the 64-bit addressing mode and does not contain a G-type instruction, it cannot change bits 0-31 of any general register.

Input/Output

Additional I/O functions and facilities are provided when z/Architecture is installed. They are provided in both the ESA/390 and the z/Architecture architectural mode and are as follows:

- Indirect data addressing is enhanced by the provision of a doubleword format-2 IDAW that is intended to allow operations on data at or above the 2G-byte absolute-address boundary in z/Architecture. The previously existing IDAW, a word containing a 31-bit address, is now called a format-1 IDAW. The format-2 IDAW contains a 64-bit address. A bit in the operation-request block (ORB) associated with a channel program specifies whether the program uses format-1 or format-2 IDAWs. A further enhancement is the ability of all format-2 IDAWs of a channel program to specify either 2K-byte or 4K-byte data blocks, as determined by another bit in the ORB. The use of 4K-byte blocks improves the efficiency of data transfers.
- The FICON-channel facility provides the capabilities of attaching FICON-I/O-interface and FICON-converted-I/O-interface channel paths
and of fully utilizing these channel-path types. FICON channel paths can significantly enhance overall data throughput by providing increased data-transfer rates in comparison to ESCON channel paths and by allowing multiple commands and associated data to be “streamed” to control units, thus further improving performance. The facility supports the following additional control mechanisms:

- The modification-control bit in the ORB allows the program to optimize the performance of FICON channel paths when dynamically modifying channel programs.
- The synchronization-control bit in the ORB ensures data integrity along with maximum channel-path performance by delaying the execution of a write command until the completion of an immediately preceding read command when performing unlimited prefetching of CCWs and when the data to be written may be the data read.
- The streaming-mode-control bit in the ORB allows the program to prevent command streaming in cases that require such prevention.
- The secondary-CCW-address field in the extended-status word assists in the recovery of channel programs that terminate abnormally when command streaming to a control unit is being performed. The field identifies a CCW that failed at the control unit.

- The ORB-extension facility expands the size of the ORB from three words to eight words. This makes fields available for use by the channel-subsystem-I/O-priority facility.
- The channel-subsystem-I/O-priority facility allows the program to establish a priority relationship among subchannels that have pending I/O operations. The priority relationship specifies the order in which I/O operations are initiated by the channel subsystem. Additionally, for fibre-channel-attached control units, the facility allows the program to specify the priority in which I/O operations pending at the control unit are performed.

The input/output enhancements are further highlighted below by describing how they affect the I/O chapters.

- In Chapter 13, “I/O Overview,” FICON and FICON-converted I/O interfaces and the frame-multiplex mode are introduced.
- In Chapter 14, “I/O Instructions”:
  - The CANCEL SUBCHANNEL instruction is described.
  - TEST PENDING INTERRUPTION, when the second-operand address is zero, stores a three-word I/O-interruption code at real locations 184-195. The new third word contains an interruption-identification word that further identifies the source of the I/O interruption.
- In Chapter 15, “Basic I/O Functions”:
  - The ORB is extended to eight words and newly contains a streaming-mode control, modification control, synchronization control, format-2-IDA control, 2K-IDA control, ORB-extension control, channel-subsystem priority, and control-unit priority.
  - A doubleword format-2 IDA and 4K-byte data blocks optionally designated by format-2 IDAWs are added.
- In Chapter 16, “I/O Interruptions”:
  - A secondary-CCW-address-validity bit and failing-storage-address-format bit are added to the extended-report word.
  - A two-word failing-storage address and a secondary-CCW address are added to the format-0 extended-status word.
- In Chapter 17, “I/O Support Functions”:
  - Control-unit-defer time is added. This has an effect on the device-connect time and device-disconnect time in the measurement block.
  - References to the measurement block by the measurement-block-update facility are single-access references and appear to be word concurrent as observed by CPUs.
  - Device-active-only time is added to the measurement block.
  - The channel-subsystem-I/O-priority facility, providing channel-subsystem priority and control-unit priority, is added.
Additions to z/Architecture

z/Architecture was announced in October, 2000. Any extension added subsequently is summarized below and has the date of its announcement at the end of its summary.

Extended-Translation Facility 2

The extended-translation facility 2 may be available on a model implementing z/Architecture. The facility performs operations on double-byte, ASCII, and decimal data. The double-byte data may be Unicode™ data — data that uses the binary codes of the Unicode Worldwide Character Standard and enables the use of characters of most of the world's written languages. The facility provides the following instructions:

- COMPARE LOGICAL LONG UNICODE
- MOVE LONG UNICODE
- PACK ASCII
- PACK UNICODE
- TEST DECIMAL
- TRANSLATE ONE TO ONE
- TRANSLATE ONE TO TWO
- TRANSLATE TWO TO ONE
- TRANSLATE TWO TO TWO
- UNPACK ASCII
- UNPACK UNICODE

The extended-translation facility 2 is called facility 2 since an extended-translation facility, now called facility 1, was introduced in ESA/390. Facility 1 is standard in z/Architecture. Facility 1 provides the instructions:

- CONVERT UNICODE TO UTF-8
- CONVERT UTF-8 TO UNICODE
- TRANSLATE EXTENDED

For when either or both of facility 1 and facility 2 are not installed on the machine, both facilities are simulated by the MVS CSRUNIC macro instruction, which is provided in OS/390* Release 10 and z/OS*.

HFP Multiply-and-Add/Subtract Facility

The HFP-multiply-add/subtract facility provides instructions for improved processing of hexadecimal floating-point numbers. The MUL-

TIPLY AND ADD (or SUBTRACT) instruction is intended to be used in place of MULTIPLY followed by ADD (or SUBTRACT) NORMALIZED.

Message-Security Assist

The message-security assist (MSA) may be available on a model implementing z/Architecture. The MSA basic facility includes the following instructions:

- CIPHER MESSAGE
- CIPHER MESSAGE WITH CHAINING
- COMPUTE INTERMEDIATE MESSAGE DIGEST
- COMPUTE LAST MESSAGE DIGEST
- COMPUTE MESSAGE AUTHENTICATION CODE

Also included are five query functions and two functions for generating a message digest based on the secure-hash algorithm (SHA-1). The five query functions, one for each instruction, are used to determine the additional installed MSA facilities, which may include the following.

MSA Data-Encryption-Algorithm (DEA) Facility:
The MSA DEA facility consists of nine functions for ciphering messages, with or without chaining, and for generating a message-authentication code (MAC) using a 56-bit, 112-bit, or 168-bit
cryptographic key. All of these functions are based on the DEA algorithm.

(June, 2003)

Long-Displacement Facility

The long-displacement facility provides a 20-bit signed-displacement field in 69 previously existing instructions (by using a previously unused byte in the instructions) and 44 new instructions. A 20-bit signed displacement allows relative addressing of up to 524,287 bytes beyond the location designated by a base register or base-and-index-register pair and up to 524,288 bytes before that location. The enhanced previously existing instructions generally are ones that handle 64-bit binary integers. The new instructions generally are new versions of instructions for 32-bit binary integers. The new instructions also include (1) a LOAD BYTE instruction that sign-extends a byte from storage to form a 32-bit or 64-bit result in a general register and (2) new floating-point LOAD and STORE instructions. The long-displacement facility provides register-constraint relief by reducing the need for base registers, code size reduction by allowing fewer instructions to be used, and additional improved performance through removal of possible address-generation interlocks. (June, 2003)

Extended-I/O-Measurement-Block Facility

The extended-I/O-measurement-block facility may be available on models implementing z/Architecture. The facility includes the following features:

- A new format of the channel-measurement block. The new measurement block, termed a format-1 channel-measurement block, is expanded to 64 bytes and is addressed using a separate measurement-block address for each subchannel. The new measurement-block format provides additional measurement information and the flexibility to store the measurement blocks in non-contiguous, real storage.
- The previously existing channel-measurement block is termed a format-0 channel-measurement block. A device-busy-time field is added to the format-0 channel-measurement block.

(June, 2003)

Extended-I/O-Measurement-Word Facility

The extended-I/O-measurement-word facility may be available on models implementing z/Architecture. The extended-measurement-word (EMW) is an extension to the interruption-response block (IRB) and allows channel-measurement data to be provided on an I/O operation basis. This reduces program overhead by alleviating the previous requirement that the program fetch the measurement block before and after an operation and calculate the difference between the respective measurement data values.

(June, 2003)

Extended-Translation Facility 3

The extended-translation facility 3 may be available on a model implementing z/Architecture. The facility performs operations on Unicode and Unicode-transformation-format (UTF) characters; it also includes a right-to-left TRANSLATE AND TEST operation. The facility provides the following instructions:

- CONVERT UTF-16 TO UTF-32
- CONVERT UTF-32 TO UTF-16
- CONVERT UTF-32 TO UTF-8
- CONVERT UTF-8 TO UTF-32
- SEARCH STRING UNICODE
- TRANSLATE AND TEST REVERSED

(May, 2004)

---

1 These key lengths reflect the cryptographic strength. In subsequent chapters, they are referred to as 64-bit, 128-bit, or 192-bit, respectively, to include the DEA-key-parity bits.
ASN-and-LX-Reuse Facility

The ASN-and-LX-reuse facility may be available on a model implementing z/Architecture. The facility provides the means by which an address-space number (ASN) that is used in certain space-switching linkage instructions may be safely reused. The facility also adds a 32-bit program-call (PC) number, and it also provides the means by which the linkage index that is used in PC-number translation may be safely reused. The facility provides the following instructions:

- EXTRACT PRIMARY ASN AND INSTANCE
- EXTRACT SECONDARY ASN AND INSTANCE
- PROGRAM TRANSFER WITH INSTANCE
- SET SECONDARY ASN WITH INSTANCE

(May, 2004)

List-Directed Initial Program Load

The list-directed initial-program-load (IPL) function may be available on a model. List-directed IPL, also known as SCSI IPL in other System Library publications, provides the means by which a program can be loaded from an I/O device other than a classical channel-attached device (ESCON or FICON). Facilities are also provided for loading a stand-alone dump program using list-directed IPL.

(May, 2004)

The ESA/390 Base

z/Architecture includes all of the facilities of ESA/390 except for the asynchronous-pageout, asynchronous-data-mover, program-call-fast, and vector facilities. This section briefly outlines most of the remaining facilities that were additions in ESA/390 as compared to ESA/370.

ESA/390 is described in Enterprise Systems Architecture/390 Principles of Operation, SA22-7201.

The CPU-related facilities that were new in ESA/390 are summarized below. ESA/390 was announced in September, 1990. Any extension added subsequently has the date of its announcement in parentheses at the end of its summary.

The following extensions are described in detail in SA22-7201 and in this publication:

- **Access-list-controlled protection** allows store-type storage references to an address space to be prohibited by means of a bit in the access-list entry used to access the space. Thus, different users having different access lists can have different capabilities to store in the same address space.

- **The program-event-recording facility 2 (PER 2)** is an alternative to the original PER facility, which is now named PER 1. (Neither of the names PER 1 and PER 2 is used in z/Architecture; only “PER” is used.) PER 2 provides the option of having a successful-branching event occur only when the branch target is within the designated storage area, and it provides the option of having a storage-alteration event occur only when the storage area is within designated address spaces. The use of these options improves performance by allowing only PER events of interest to occur. PER 2 deletes the ability to monitor for general-register-alteration events.

- **Concurrent sense** improves performance by allowing sense information to be presented at the time of an interruption due to a unit-check condition, thus avoiding the need for a separate I/O operation to obtain the sense information.

- **Broadcasted purging** provides the COMPARE AND SWAP AND PURGE instruction for conditionally updating tables associated with dynamic address translation and access-register translation and clearing associated buffers in multiple CPUs. This is described in detail beginning in the eighth edition of SA22-7201.

- **Storage-protection override** provides a new form of subsystem storage protection that improves the reliability of a subsystem executed in an address space along with possibly erroneous application programs. When storage-protection override is made active by a control-register bit, fetches and stores by the CPU are permitted to storage locations having a storage key of 9 regardless of the access
key used by the CPU. If the subsystem is in key-8 storage and is executed with a PSW key of 8, for example, and the application programs are in key-9 storage and are executed with a PSW key of 9, accesses by the subsystem to the application-program areas are permitted while accesses by the application programs to the subsystem area are denied. (September, 1991)

**Move-page facility 2** extends the MOVE PAGE instruction introduced in ESA/370 by allowing use of a specified access key for either the source or the destination operand, by allowing improved performance when the destination operand will soon be referenced, and by allowing improved performance when an operand is invalid in both main and expanded storage. The ESA/370 version of MOVE PAGE is now called move-page facility 1 and is in Chapter 7, “General Instructions.” MOVE PAGE of move-page facility 2 is in Chapter 10, “Control Instructions.” Some details about the means for control-program support of MOVE PAGE are not provided. (September, 1991) (The z/Architecture MOVE PAGE instruction is described only in Chapter 10 of this publication. MOVE PAGE no longer can move data to or from expanded storage, and all details about MOVE PAGE are provided.)

**The square-root facility** consists of the SQUARE ROOT instruction and the square-root exception. The instruction extracts the square root of a floating-point operand in either the long or short format. The instruction is the same as that provided on some models of the IBM 4341, 4361, and 4381 Processors. (September, 1991)

**The cancel-I/O facility** allows the program to withdraw a pending start function from a designated subchannel without signaling the device, which is useful in certain error-recovery situations. (September, 1991)

The cancel-I/O facility provides the CANCEL SUBCHANNEL instruction and is described in detail beginning in the eighth edition of SA22-7201.

**The string-instruction facility** (or logical string assist) provides instructions for (1) moving a string of bytes until a specified ending byte is found, (2) logically comparing two strings until an inequality or a specified ending byte is found, and (3) searching a string of a specified length for a specified byte. The first two instructions are particularly useful in a C program in which strings are normally delimited by an ending byte of all zeros. (June, 1992)

**The suppression-on-protection facility** causes a protection exception due to page protection to result in suppression of instruction execution instead of termination of instruction execution, and it causes the address and an address-space identifier of the protected page to be stored in low storage. This is useful in performing the AIX/ESA* copy-on-write function, in which AIX/ESA causes the same page of different address spaces to map to a single page frame of real storage so long as a store in the page is not attempted and then, when a store is attempted in a particular address space, assigns a unique page frame to the page in that address space and copies the contents of the page to the new page frame. (February, 1993)

**The set-address-space-control-fast facility** consists of the SET ADDRESS SPACE CONTROL FAST (SACF) instruction, which possibly can be used instead of the previously existing SET ADDRESS SPACE CONTROL (SAC) instruction, depending on whether all of the SAC functions are required. SACF, unlike SAC, does not perform the serialization and checkpoint-synchronization functions, nor does it cause copies of prefetched instructions to be discarded. SACF provides improved performance on some models. (February, 1993)

**The subspace-group facility** includes the BRANCH IN SUBSPACE GROUP instruction, which can be used to give or return control from one address space to another in a group of address spaces called a subspace group, with this giving and returning of control being done with better performance than can be obtained by means of the PROGRAM CALL and PROGRAM RETURN or PROGRAM TRANSFER instructions. One address space in the subspace group is called the base

---

AIX/ESA, CICS, and MVS/ESA are trademarks of the International Business Machines Corporation.
space, and the other address spaces in the group are called subspaces. It is intended that each subspace contain a different subset of the storage in the base space, that the base space and each subspace contain a subsystem control program, such as CICS*, and application programs, and that each subspace contain the data for a single transaction being processed under the subsystem control program. The placement of the data for each transaction in a different subspace prevents the processing of a transaction from erroneously damaging the data of other transactions. The data of the control program can be protected from the transaction processing by means of the storage-protection-override facility. (April, 1994)

- The **virtual-address enhancement of suppression on protection** provides that if dynamic address translation (DAT) was on when a protection exception was recognized, the suppression-on-protection result is indicated, and the address of the protected location is stored, only if the address is one that was to be translated by DAT; the suppression-on-protection result is not indicated if the address that would be stored is a real address. This enhancement allows the stored address to be translated reliably by the control program to determine if the exception was due to page protection as opposed to key-controlled protection. The enhancement extends the usefulness of suppression on protection to operating systems like MVS/ESA* that use key-controlled protection. (September, 1994)

- The **immediate-and-relative-instruction facility** includes 13 new instructions, most of which use a halfword-immediate value for either signed-binary arithmetic operations or relative branching. The facility reduces the need for general registers, and, in particular, it eliminates the need to use general registers to address branch targets. As a result, the general registers and access registers can be allocated more efficiently in programs that require many registers. (September, 1996)

- The **compare-and-move-extended facility** provides new versions of the COMPARE LOGICAL LONG and MOVE LONG instructions. The new versions increase the size of the operand-length specifications from 24 bits to 32 bits, which can be useful when objects larger than 16M bytes are processed through the use of 31-bit addressing. The new versions also periodically complete to allow software polling in a multiprocessing system. (September, 1996)

- The **checksum facility** consists of the CHECKSUM instruction, which can be used to compute a 16-bit or 32-bit checksum in order to improve TCP/IP (transmission-control protocol/internet protocol) performance. (September, 1996)

- The **called-space-identification facility** improves serviceability by further identifying the called address space in a linkage-stack state entry formed by the PROGRAM CALL instruction. (September, 1996)

- The **branch-and-set-authority facility** consists of the BRANCH AND SET AUTHORITY instruction, which can be used to improve the performance of linkages within an address space by replacing PROGRAM CALL, PROGRAM TRANSFER, and SET PSW KEY FROM ADDRESS instructions. (June, 1997)

- The **perform-locked-operation facility** consists of the unprivileged PERFORM LOCKED OPERATION instruction, which appears to provide concurrent interlocked-update references to multiple storage operands. A function code of the instruction can specify any of six operations: compare and load, compare and swap, double compare and swap, compare and swap and store, compare and swap and double store, and compare and swap and triple store. The function code further specifies either word or doubleword operands. The instruction can be used to avoid the use of programmed locks in a multiprocessing system. (June, 1997)

- Four additional floating-point facilities improve the hexadecimal-floating-point (HFP) capability of the machine and add a binary-floating-point (BFP) capability. The facilities are:
  - Basic floating-point extensions, which provides 12 additional floating-point registers to make a total of 16 floating-point registers. This facility also includes a floating-point-control register and means for saving the contents of the new registers during a store-status operation or a machine-check interruption.
Floating-point-support (FPS) extensions, which provides eight new instructions, including four to convert data between the HFP and BFP formats.

Hexadecimal-floating-point (HFP) extensions, which provides 26 new instructions to operate on data in the HFP format. All of these are counterparts to new instructions provided by the BFP facility, including conversion between floating-point and fixed-point formats, and a more complete set of operations on the extended format.

Binary floating-point (BFP), which defines short, long, and extended binary-floating-point (BFP) data formats and provides 87 new instructions to operate on data in these formats. The BFP formats and operations provide everything necessary to conform to the IEEE standard (ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, dated August 12, 1985) except for conversion between binary-floating-point numbers and decimal strings, which must be provided in software. (May, 1998)

• The resume-program facility consists of the RESUME PROGRAM instruction, which restores, from a specified save area, the instruction address and certain other fields in the current PSW and also the contents of an access-and-general-register pair. RESUME PROGRAM allows a problem-state interruption-handling program to restore the state of an interrupted program and return to that program despite that a register is required for addressing the save area from which the state is restored. (May, 1998)

• The trap facility provides the TRAP instructions (a two-byte TRAP2 instruction and a four-byte TRAP4 instruction) that can overlay instructions in an application program to give control to a program that can perform fix-up operations on data being processed, such as dates that may be a “Year-2000” problem. RESUME PROGRAM can be used to return from the fix-up program. TRAP and RESUME PROGRAM can improve performance by avoiding program interruptions that would otherwise be needed to give control to and from the fix-up program. (May, 1998)

• The extended-TOD-clock facility includes (1) an extension of the TOD clock from 64 bits to 104 bits, allowing greater resolution; (2) a TOD programmable register, which contains a TOD programmable field that can be used to identify the configuration providing a TOD-clock value in a sysplex; (3) the SET CLOCK PROGRAMMABLE FIELD instruction, for setting the TOD programmable field in the TOD programmable register; and (4) the STORE CLOCK EXTENDED instruction, which stores both the longer TOD-clock value and the TOD programmable field. STORE CLOCK EXTENDED can be used in the future when the TOD clock is further extended to contain time values that exceed the current year-2042 limit (when there is a carry out of the current bit 0 of the TOD clock). (August, 1998)

• The TOD-clock-control-override facility provides a control-register bit that allows setting the TOD clock under program control, without use of the manual TOD-clock control of any CPU. (August, 1998)

• The store-system-information facility provides the privileged STORE SYSTEM INFORMATION instruction, which can be used to obtain information about a component or components of a virtual machine, a logical partition, or the basic machine. (January, 1999)

• The extended-translation facility, now called the extended-translation facility 1, includes the CONVERT UNICODE TO UTF-8, CONVERT UTF-8 TO UNICODE, and TRANSLATE EXTENDED instructions, all of which can improve performance. TRANSLATE EXTENDED can be used in place of a TRANSLATE AND TEST instruction that locates an escape character, followed by a TRANSLATE instruction that translates the bytes preceding the escape character. (April, 1999)

The following extensions are described in detail in other publications:

• The Enterprise Systems Connection Architecture (ESCON) introduces a new type of channel that uses an optical-fiber communication link between channels and control units. Information is transferred serially by bit,
at 200 million bits per second, up to a maximum distance of 60 kilometers. The optical-fiber technology and serial transmission simplify cabling and improve reliability. See the publication *IBM Enterprise Systems Architecture/390 ESCON I/O Interface*, SA22-7202.

- The **ESCON channel-to-channel adapter (ESCON CTCA)** provides the same type of function for serial channel paths as is available for the parallel-I/O-interface channel paths. See the publication *IBM Enterprise Systems Architecture/390 ESCON Channel-to-Channel Adapter*, SA22-7203.

- **I/O-device self-description** allows a device to describe itself and its position in the I/O configuration. See the publication *IBM Enterprise Systems Architecture/390 Common I/O-Device Commands and Self Description*, SA22-7204.

- The **compression facility** performs a Ziv-Lempel type of compression and expansion by means of static (nonadaptive) dictionaries that are to be prepared by a program before the compression and expansion operations. Because the dictionaries are static, the compression facility can provide good compression not only for long sequential data streams (for example, archival or network data) but also for randomly accessed short records (for example, 80 bytes). See the publication *IBM Enterprise Systems Architecture/390 Data Compression*, SA22-7208. (February, 1993) (The *z/Architecture COMPRESSION CALL instruction* is described in this publication. However, introductory information and information about dictionary formats still is provided only in SA22-7208.)

The remaining extensions of ESA/390, for which detailed descriptions are not provided, are as follows:

- The integrated **cryptographic facility** provides a number of instructions to protect data privacy, to support message authentication and personal identification, and to facilitate key management. The high-performance cipher capability of the facility is designed for financial-transaction and bulk-encryption environments, and it complies with the Data Encryption Standard (DES).

- **Usability of the cryptographic facility** is extended to virtual-machine environments, which allows the facility to be used by MVS/ESA being executed under VM/ESA*, which in turn may be executed either under another VM/ESA or in a logical partition. (September, 1991)

- The **external-time-reference facility** provides a means to initiate and maintain the synchronization of TOD clocks to an external time reference (ETR). Synchronization tolerance of a few microseconds can be achieved, and the effect of leap seconds is taken into account. The facility consists of an ETR sending unit (Sysplex Timer*), which may be duplexed, two or more ETR receiving units, and optical-fiber cables. The cables are used to connect the ETR sending unit, which is an external device, to ETR receiving units of the configuration. CPU instructions are provided for setting the TOD clock to the value supplied by the ETR sending unit.

  - The ETR **automatic-propagation-delay-adjustment** function adjusts the time signals from the ETR to the attached processors to compensate for the propagation delay on the cables to the processors, thus allowing the cables to be of different lengths. (September, 1991)

  - The ETR **external-time-source** function synchronizes the ETR to a time signal received from a remote location by means of a telephone or radio. (September, 1991)

- **Extended sorting** provides instructions that improve the performance of the DB2* sorting function.

- Other **PER extensions**, besides those described beginning in the fourth edition of this publication, are an augmentation of PER 2 that provide additional PER function in the interpretive-execution mode.

- **Channel-subsystem call** provides various functions for use in the management of the I/O configuration. Some of the functions acquire

---

MVS/ESA, VM/ESA, Sysplex Timer, and DB2 are trademarks of the International Business Machines Corporation.
information about the configuration from the accessible elements of the configuration, while others dynamically change the configuration.

- The operational extensions are a number of other improvements that result in increased availability and ease of use of the system, as follows:
  
  - **Automatic-reconfiguration** permits an operating system in an LPAR partition to declare itself willing to be terminated suddenly, usually to permit its storage and CPU resources to be acquired by an adjacent partition that is dynamically absorbing the workload of another system that has failed. Other functions deactivate and reset designated participating partitions.

  - A new **storage-reconfiguration command** decreases the time needed to reconfigure storage by allowing multiple requests for reconfiguration to be made by means of a single communication with the service processor.

  - **SCP-initiated reset** allows a system control program (SCP) to reset its I/O configuration prior to entering the disabled wait state following certain check conditions.

  - **Console integration** simplifies configuration requirements by reducing by one the number of consoles required by MVS.

  - The **processor-availability facility** enables a CPU experiencing an unrecoverable error that will cause a check stop to save its state and alert the other CPUs in the configuration. This allows, in many cases, another CPU to continue execution of the program that was in execution on the failing CPU. The facility is applicable in both the ESA/390 mode and the LPAR mode. (April, 1991)

- **Extensions for virtual machines** are a number of improvements to the interpretive-execution facility, as follows:

  - The **VM-data-space facility** provides for making the ESA/390 access-register architecture more useful in virtual-machine applications. The facility improves the ability to address a larger amount of data and to share data. For information on how VM/ESA uses the VM-data-space facility, see the publication VM/ESA CP Programming Services, SC24-5520.

  - A new **storage-key function** improves performance by removing the need for the previously used RCP area.

  - Interpreted SIE (available with region relocation) is improved to permit preferred guests under VM when VM itself is operating as a high-performance guest.

  - Other improvements include an optional special-purpose lookaside for some of the guest-state information and greater freedom in certain implementation choices.

- The **ESCON-multiple-image facility (EMIF)** allows multiple logical partitions to share ESCON channels (and FICON channels) and optionally to share any of the control units and associated I/O devices configured to these shared channels. This can reduce channel requirements, improve channel utilization, and improve I/O connectivity. (June, 1992)

- **PR/SM LPAR mode** is enhanced to allow up to 10 logical partitions in a single-image configuration and 20 in a physically-partitioned configuration. The previous limits were seven and 14, respectively. (June, 1992)

  Coincident with z/Architecture, PR/SM LPAR mode allows 15 logical partitions, and physical partitioning is not supported.

- The **coupling facility** enables high-performance data sharing among MVS/ESA systems that are connected by means of the facility. The coupling facility provides storage that can be dynamically partitioned for caching data in shared buffers, maintaining work queues and status information in shared lists, and locking data by means of shared lock controls. MVS/ESA services provide access to and manipulation of the coupling-facility contents. (April, 1994)

**The ESA/370 and 370-XA Base**

ESA/390 includes the complete set of facilities of ESA/370 as its base. This section briefly outlines most of the facilities that were additions in 370-XA as compared to System/370 and that were additions in ESA/370 as compared to 370-XA.
The CPU-related facilities that were new in 370-XA are as follows:

- **Bimodal addressing** provides two modes of operation: a 24-bit addressing mode for the execution of old programs and a 31-bit addressing mode.
- **31-bit logical addressing** extends the virtual address space from the 16M bytes addressable with 24-bit addresses to 2G bytes (2,147,483,648 bytes).
- **31-bit real and absolute addressing** provides addressability for up to 2G bytes of main storage.

The 370-XA protection facilities include key-controlled protection on only 4K-byte blocks, page protection, and, as in System/370, low-address protection for addresses below 512. Fetch-protection override eliminates fetch protection for locations 0-2047.

- **The tracing facility** assists in the determination of system problems by providing an ongoing record in storage of significant events.
- The COMPARE AND FORM CODEWORD and UPDATE TREE instructions facilitate sorting applications.
- The interpretive-execution facility allows creation of virtual machines that may operate according to several architectures and whose performance is enhanced because many virtual-machine functions are directly interpreted by the machine rather than simulated by the program. This facility is described in the publication *IBM 370-XA Interpretive Execution*, SA22-7095.
- The service-call-logical-processor (SCLP) facility provides a means of communicating between the control program and the service processor for the purpose of describing and changing the configuration. This facility is not described.

The I/O-related differences between 370-XA and System/370 result from the 370-XA channel subsystem, which includes:

- **Path-independent addressing** of I/O devices, which permits the initiation of I/O operations without regard to which CPU is executing the I/O instruction or how the I/O device is attached to the channel subsystem. Any I/O interruption can be handled by any CPU enabled for it.
- **Path management**, whereby the channel subsystem determines which paths are available for selection, chooses a path, and manages any busy conditions encountered while attempting to initiate I/O processing with the associated devices.
- **Dynamic reconnection**, which permits any I/O device using this capability to reconnect to any available channel path to which it has access in order to continue execution of a chain of commands.
- **Programmable interruption subclasses**, which permit the programmed assignment of I/O-interruption requests from individual I/O devices to any one of eight maskable interruption queues.
- **An additional CCW format** for the direct use of 31-bit addresses in channel programs. The new CCW format, called format 1, is provided in addition to the System/370 CCW format, now called format 0.
- **Address-limit checking**, which provides an additional storage-protection facility to prevent data access to storage locations above or below a specified absolute address.
- **Monitoring facilities**, which can be invoked by the program to cause the channel subsystem to measure and accumulate key I/O-resource usage parameters.
- **Status-verification facility**, which reports inappropriate combinations of device-status bits presented by a device.
- A set of 13 I/O instructions, with associated control blocks, which are provided for the control of the channel subsystem.

The facilities that were new in ESA/370 are as follows:

- Sixteen access registers permit the program to have immediate access to storage operands in up to 16 2G-byte address spaces, including the address space in which the program resides. In a dynamic-address-translation mode named access-register mode, the instruction B field, or for certain instructions the R field, designates both a general register and an access register, and the contents of the access register,
along with the contents of protected tables, specify the operand address space to be accessed. By changing the contents of the access registers, the program, under the control of an authorization mechanism, can have fast access to hundreds of different operand address spaces.

- A linkage stack is used in a functionally expanded mechanism for passing control between programs in either the same or different address spaces. This mechanism makes use also of the previously existing PROGRAM CALL instruction, an extended entry-table entry, and a new PROGRAM RETURN instruction. The mechanism saves various elements of status, including access-register and general-register contents, during a calling linkage, provides for changing the current status during the calling linkage, and restores the original status during the returning linkage. The linkage stack can also be used to save and restore access-register and general-register contents during a branch-type linkage performed by the new instruction BRANCH AND STACK.

- A translation mode named home-space mode provides an efficient means for the control program to obtain control in the address space, called the home address space, in which the principal control blocks for a dispatchable unit (a task or process) are kept.

- The semiprivileged MOVE WITH SOURCE KEY and MOVE WITH DESTINATION KEY instructions allow bidirectional movement of data between storage areas having different storage keys, without the need to change the PSW key.

- The privileged LOAD USING REAL ADDRESS and STORE USING REAL ADDRESS instructions allow the control program to access data in real storage more efficiently.

- The private-space facility allows an address space not to contain any common segments and causes low-address protection and fetch-protection override not to apply to the address space.

- The unprivileged MOVE PAGE instruction allows the program to move a page of data between main and expanded storage, provided that the source and destination pages are both valid. Some details about the means for control-program support of MOVE PAGE are not provided. The ESA/370 version of MOVE PAGE is now called move-page facility 1.

- The Processor Resource/Systems Manager* (PR/SM*) feature provides support for multiple preferred guests under VM/XA and provides the logically partitioned (LPAR) mode, with the latter providing flexible partitioning of processor resources among multiple logical partitions. Certain aspects of the LPAR use of PR/SM are described in the publication IBM ES/3090 Processor Complex Processor Resource/Systems Manager Planning Guide, GA22-7123.

- The COMPARE UNTIL SUBSTRING EQUAL instruction provides improved performance of the compression of IMS log data sets and can be useful in other programs also.

System Program

z/Architecture is designed to be used with a control program that coordinates the use of system resources and executes all I/O instructions, handles exceptional conditions, and supervises scheduling and execution of multiple programs.

Compatibility

Compatibility among z/Architecture Systems

Although systems operating as defined by z/Architecture may differ in implementation and physical capabilities, logically they are upward and downward compatible. Compatibility provides for simplicity in education, availability of system backup, and ease in system growth. Specifically, any program written for z/Architecture gives iden-
tical results on any z/Architecture implementation, provided that the program:

1. Is not time-dependent.
2. Does not depend on system facilities (such as storage capacity, I/O equipment, or optional facilities) being present when the facilities are not included in the configuration.
3. Does not depend on system facilities being absent when the facilities are included in the configuration. For example, the program must not depend on interruptions caused by the use of operation codes or command codes that are not installed in some models. Also, it must not use or depend on fields associated with uninstalled facilities. For example, data should not be placed in an area used by another model for fixed-logoff information. Similarly, the program must not use or depend on unassigned fields in machine formats (control registers, instruction formats, etc.) that are not explicitly made available for program use.
4. Does not depend on results or functions that are defined to be unpredictable or model-dependent or are identified as undefined. This includes the requirement that the program should not depend on the assignment of device numbers and CPU addresses.
5. Does not depend on results or functions that are defined in the functional-characteristics publication for a particular model to be deviations from the architecture.
6. Takes into account any changes made to the architecture that are identified as affecting compatibility.

Compatibility between z/Architecture and ESA/390

Problem-State Compatibility
A high degree of compatibility exists at the problem-state level in going forward from ESA/390 to z/Architecture. Because the majority of a user’s applications are written for the problem state, this problem-state compatibility is useful in many installations.

A problem-state program written for ESA/390 operates with z/Architecture, provided that the program:

1. Complies with the limitations described in “Compatibility among z/Architecture Systems” on page 1-16.
2. Is not dependent on privileged facilities which are unavailable on the system.

Programming Note: This publication assigns meanings to various operation codes, to bit positions in instructions, channel-command words, registers, and table entries, and to fixed locations in the low 512 bytes and bytes 4096-8191 of storage. Unless specifically noted, the remaining operation codes, bit positions, and low-storage locations are reserved for future assignment to new facilities and other extensions of the architecture.

To ensure that existing programs operate if and when such new facilities are installed, programs should not depend on an indication of an exception as a result of invalid values that are currently defined as being checked. If a value must be placed in unassigned positions that are not checked, the program should enter zeros. When the machine provides a code or field, the program should take into account that new codes and bits may be assigned in the future. The program should not use unassigned low-storage locations for keeping information since these locations may be assigned in the future in such a way that the machine causes the contents of the locations to be changed.

Availability
Availability is the capability of a system to accept and successfully process an individual job. Systems operating in accordance with z/Architecture permit substantial availability by (1) allowing a large number and broad range of jobs to be processed concurrently, thus making the system readily accessible to any particular job,
and (2) limiting the effect of an error and identifying more precisely its cause, with the result that the number of jobs affected by errors is minimized and the correction of the errors facilitated.

Several design aspects make this possible.

- A program is checked for the correctness of instructions and data as the program is executed, and program errors are indicated separate from equipment errors. Such checking and reporting assists in locating failures and isolating effects.
- The protection facilities, in conjunction with dynamic address translation and the separation of programs and data in different address spaces, permit the protection of the contents of storage from destruction or misuse caused by erroneous or unauthorized storing or fetching by a program. This provides increased security for the user, thus permitting applications with different security requirements to be processed concurrently with other applications.
- Dynamic address translation allows isolation of one application from another, still permitting them to share common resources. Also, it permits the implementation of virtual machines, which may be used in the design and testing of new versions of operating systems along with the concurrent processing of application programs. Additionally, it provides for the concurrent operation of incompatible operating systems.
- The use of access registers allows programs, data, and different collections of data to reside in different address spaces, and this further reduces the likelihood that a store using an incorrect address will produce either erroneous results or a system-wide failure.
- Multiprocessing and the channel subsystem permit better use of storage and processing capabilities, more direct communication between CPUs, and duplication of resources, thus aiding in the continuation of system operation in the event of machine failures.
- MONITOR CALL, program-event recording, and the timing facilities permit the testing and debugging of programs without manual intervention and with little effect on the concurrent processing of other programs.
- On most models, error checking and correction (ECC) in main storage, CPU retry, and command retry provide for circumventing intermittent equipment malfunctions, thus reducing the number of equipment failures.
- An enhanced machine-check-handling mechanism provides model-independent fault isolation, which reduces the number of programs impacted by uncorrected errors. Additionally, it provides model-independent recording of machine-status information. This leads to greater machine-check-handling compatibility between models and improves the capability for loading and operating a program on a different model when a system failure occurs.
- A small number of manual controls are required for basic system operation, permitting most operator-system interaction to take place via a unit operating as an I/O device and thus reducing the possibility of operator errors.
- The logical partitions made available by the PR/SM feature allow continued reliable production operations in one or more partitions while new programming systems are tested in other partitions. This is an advancement in particular for non-VM installations.
- The operational extensions and channel-subsystem-call facility of ESA/390 and z/Architecture improve the ability to continue execution of application programs in the presence of system incidents and the ability to make configuration changes with less disruption to operations.
Chapter 2. Organization

Main Storage . . . . . . . . . . . . . . . . . . . 2-2
Expanded Storage . . . . . . . . . . . . . . . 2-2
CPU . . . . . . . . . . . . . . . . . . . . . . . 2-2
PSW . . . . . . . . . . . . . . . . . . . . . . 2-3
General Registers . . . . . . . . . . . . . . 2-3
Floating-Point Registers . . . . . . . . . . . 2-3
Floating-Point-Control Register . . . . . . . 2-4
Control Registers . . . . . . . . . . . . . . . 2-4
Access Registers . . . . . . . . . . . . . . . 2-4
Cryptographic Facility . . . . . . . . . . . . 2-6
External Time Reference . . . . . . . . . . . 2-6
I/O . . . . . . . . . . . . . . . . . . . . . . . . 2-6
Channel Subsystem . . . . . . . . . . . . . 2-6
Channel Paths . . . . . . . . . . . . . . . . . 2-6
I/O Devices and Control Units . . . . . . . 2-7
Operator Facilities . . . . . . . . . . . . . . 2-7

Logically, a system consists of main storage, one or more central processing units (CPUs), operator facilities, a channel subsystem, and I/O devices. I/O devices are attached to the channel subsystem through control units. The connection between the channel subsystem and a control unit is called a channel path.

A channel path employs either a parallel-transmission protocol or a serial-transmission protocol and, accordingly, is called either a parallel or a serial channel path. A serial channel path may connect to a control unit through a dynamic switch that is capable of providing different internal connections between the ports of the switch.

Expanded storage may also be available in the system, a cryptographic unit may be included in a CPU, and an external time reference (ETR) may be connected to the system.

The physical identity of the above functions may vary among implementations, called “models.” Figure 2-1 depicts the logical structure of a two-CPU multiprocessing system that includes expanded storage and a cryptographic unit and that is connected to an ETR.

Specific processors may differ in their internal characteristics, the installed facilities, the number of subchannels, channel paths, and control units which can be attached to the channel subsystem, the size of main and expanded storage, and the representation of the operator facilities.

Figure 2-1. Logical Structure of a z/Architecture System with Two CPUs

A system viewed without regard to its I/O devices is referred to as a configuration. All of the physical equipment, whether in the configuration or not, is referred to as the installation.

Model-dependent reconfiguration controls may be provided to change the amount of main and
expanded storage and the number of CPUs and channel paths in the configuration. In some instances, the reconfiguration controls may be used to partition a single configuration into multiple configurations. Each of the configurations so reconfigured has the same structure, that is, main and expanded storage, one or more CPUs, and one or more subchannels and channel paths in the channel subsystem.

Each configuration is isolated in that the main and expanded storage in one configuration is not directly addressable by the CPUs and the channel subsystem of another configuration. It is, however, possible for one configuration to communicate with another by means of shared I/O devices or a channel-to-channel adapter. At any one time, the storage, CPUs, subchannels, and channel paths connected together in a system are referred to as being in the configuration. Each CPU, subchannel, channel path, main-storage location, and expanded-storage location can be in only one configuration at a time.

**Main Storage**

Main storage, which is directly addressable, provides for high-speed processing of data by the CPUs and the channel subsystem. Both data and programs must be loaded into main storage from input devices before they can be processed. The amount of main storage available in the system depends on the model, and, depending on the model, the amount in the configuration may be under control of model-dependent configuration controls. The storage is available in multiples of 4K-byte blocks. At any instant, the channel subsystem and all CPUs in the configuration have access to the same blocks of storage and refer to a particular block of main-storage locations by using the same absolute address.

Main storage may include a faster-access buffer storage, sometimes called a cache. Each CPU may have an associated cache. The effects, except on performance, of the physical construction and the use of distinct storage media are not observable by the program.

**Expanded Storage**

Expanded storage may be available on some models. Expanded storage, when available, can be accessed by all CPUs in the configuration by means of instructions that transfer 4K-byte blocks of data from expanded storage to main storage or from main storage to expanded storage. These instructions are the PAGE IN and PAGE OUT instructions, described in Chapter 10, “Control Instructions.”

Each 4K-byte block of expanded storage is addressed by means of a 32-bit unsigned binary integer called an expanded-storage block number.

**CPU**

The central processing unit (CPU) is the controlling center of the system. It contains the sequencing and processing facilities for instruction execution, interruption action, timing functions, initial program loading, and other machine-related functions.

The physical implementation of the CPU may differ among models, but the logical function remains the same. The result of executing an instruction is the same for each model, providing that the program complies with the compatibility rules.

The CPU, in executing instructions, can process binary integers and floating-point numbers (binary and hexadecimal) of fixed length, decimal integers of variable length, and logical information of either fixed or variable length. Processing may be in parallel or in series; the width of the processing elements, the multiplicity of the shifting paths, and the degree of simultaneity in performing the different types of arithmetic differ from one model of CPU to another without affecting the logical results.

Instructions which the CPU executes fall into seven classes: general, decimal, floating-point-support (FPS), binary-floating-point (BFP), hexadecimal-floating-point (HFP), control, and I/O instructions. The general instructions are used in performing binary-integer-arithmetic operations and logical, branching, and other nonarithmetic operations. The decimal instructions operate on data in the decimal format. The BFP and HFP
instructions operate on data in the BFP and HFP formats, respectively, while the FPS instructions operate on floating-point data independent of the format or convert it from one format to the other. The privileged control instructions and the I/O instructions can be executed only when the CPU is in the supervisor state; the semiprivileged control instructions can be executed in the problem state, subject to the appropriate authorization mechanisms.

The CPU provides registers which are available to programs but do not have addressable representations in main storage. They include the current program-status word (PSW), the general registers, the floating-point registers and floating-point-control register, the control registers, the access registers, the prefix register, and the registers for the clock comparator and the CPU timer. Each CPU in an installation provides access to a time-of-day (TOD) clock, which is shared by all CPUs in the installation. The instruction operation code determines which type of register is to be used in an operation. See Figure 2-2 on page 2-5 or the format of the control, access, general, and floating-point registers.

**PSW**

The program-status word (PSW) includes the instruction address, condition code, and other information used to control instruction sequencing and to determine the state of the CPU. The active or controlling PSW is called the current PSW. It governs the program currently being executed.

The CPU has an interruption capability, which permits the CPU to switch rapidly to another program in response to exceptional conditions and external stimuli. When an interruption occurs, the CPU places the current PSW in an assigned storage location, called the old-PSW location, for the particular class of interruption. The CPU fetches a new PSW from a second assigned storage location. This new PSW determines the next program to be executed. When it has finished processing the interruption, the program handling the interruption may reload the old PSW, making it again the current PSW, so that the interrupted program can continue.

There are six classes of interruption: external, I/O, machine check, program, restart, and supervisor call. Each class has a distinct pair of old-PSW and new-PSW locations permanently assigned in real storage.

**General Registers**

Instructions may designate information in one or more of 16 general registers. The general registers may be used as base-address registers and index registers in address arithmetic and as accumulators in general arithmetic and logical operations. Each register contains 64 bit positions. The general registers are identified by the numbers 0-15 and are designated by a four-bit R field in an instruction. Some instructions provide for addressing multiple general registers by having several R fields. For some instructions, the use of a specific general register is implied rather than explicitly designated by an R field of the instruction.

For some operations, either bits 32-63 or bits 0-63 of two adjacent general registers are coupled, providing a 64-bit or 128-bit format, respectively. In these operations, the program must designate an even-numbered register, which contains the leftmost (high-order) 32 or 64 bits. The next higher-numbered register contains the rightmost (low-order) 32 or 64 bits.

In addition to their use as accumulators in general arithmetic and logical operations, 15 of the 16 general registers are also used as base-address and index registers in address generation. In these cases, the registers are designated by a four-bit B field or X field in an instruction. A value of zero in the B or X field specifies that no base or index is to be applied, and, thus, general register 0 cannot be designated as containing a base address or index.

**Floating-Point Registers**

All floating-point instructions (FPS, BFP, and HFP) use the same floating-point registers. The CPU has 16 floating-point registers. The floating-point registers are identified by the numbers 0-15 and are designated by a four-bit R field in floating-point instructions. Each floating-point register is 64 bits long and can contain either a short (32-bit) or a long (64-bit) floating-point operand. As shown in Figure 2-2 on page 2-5, pairs of floating-point registers can be used for extended (128-bit) oper-
ands. Each of the eight pairs is referred to by the number of the lower-numbered register of the pair.

**Floating-Point-Control Register**

The floating-point-control (FPC) register is a 32-bit register that contains mask bits, flag bits, a data-exception code, and rounding-mode bits. The FPC register is described in the section "Floating-Point-Control (FPC) Register" on page 19-2.

**Control Registers**

The CPU has 16 control registers, each having 64 bit positions. The bit positions in the registers are assigned to particular facilities in the system, such as program-event recording, and are used either to specify that an operation can take place or to furnish special information required by the facility.

The control registers are identified by the numbers 0-15 and are designated by four-bit R fields in the instructions LOAD CONTROL and STORE CONTROL. Multiple control registers can be addressed by these instructions.

**Access Registers**

The CPU has 16 access registers numbered 0-15. An access register consists of 32 bit positions containing an indirect specification (not described here in detail) of an address-space-control element. An address-space-control element is a parameter used by the dynamic-address-translation (DAT) mechanism to translate references to a corresponding address space. When the CPU is in a mode called the access-register mode (controlled by bits in the PSW), an instruction B field, used to specify a logical address for a storage-operand reference, designates an access register, and the address-space-control element specified by the access register is used by DAT for the reference being made. For some instructions, an R field is used instead of a B field. Instructions are provided for loading and storing the contents of the access registers and for moving the contents of one access register to another.

Each of access registers 1-15 can designate any address space, including the current instruction space (the primary address space). Access register 0 always designates the current instruction space. When one of access registers 1-15 is used to designate an address space, the CPU determines which address space is designated by translating the contents of the access register. When access register 0 is used to designate an address space, the CPU treats the access register as designating the current instruction space, and it does not examine the actual contents of the access register. Therefore, the 16 access registers can designate, at any one time, the current instruction space and a maximum of 15 other spaces.
<table>
<thead>
<tr>
<th>R Field and Register Number</th>
<th>Control Registers</th>
<th>Access Registers</th>
<th>General Registers</th>
<th>Floating-Point Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note:* The arrows indicate that the two registers may be coupled as a double-register pair, designated by specifying the lower-numbered register in the R field. For example, the floating-point register pair 13 and 15 is designated by 1101 binary in the R field.

*Figure 2-2. Control, Access, General, and Floating-Point Registers*
Cryptographic Facility

Depending on the model, an integrated cryptographic facility may be provided as an extension of the CPU. When the cryptographic facility is provided on a CPU, it functions as an integral part of that CPU. A summary of the benefits of the cryptographic facility is given on page 1-13; the facility is otherwise not described.

External Time Reference

Depending on the model, an external time reference (ETR) may be connected to the configuration. A summary of the benefits of the ETR is given on page 1-13; the facility is otherwise not described.

I/O

Input/output (I/O) operations involve the transfer of information between main storage and an I/O device. I/O devices and their control units attach to the channel subsystem, which controls this data transfer.

Channel Subsystem

The channel subsystem directs the flow of information between I/O devices and main storage. It relieves CPUs of the task of communicating directly with I/O devices and permits data processing to proceed concurrently with I/O processing. The channel subsystem uses one or more channel paths as the communication link in managing the flow of information to or from I/O devices. As part of I/O processing, the channel subsystem also performs the path-management function of testing for channel-path availability, selecting an available channel path, and initiating execution of the operation with the I/O device. Within the channel subsystem are subchannels.

One subchannel is provided for and dedicated to each I/O device accessible to the channel subsystem. Each subchannel contains storage for information concerning the associated I/O device and its attachment to the channel subsystem. The subchannel also provides storage for information concerning I/O operations and other functions involving the associated I/O device. Information contained in the subchannel can be accessed by CPUs using I/O instructions as well as by the channel subsystem and serves as the means of communication between any CPU and the channel subsystem concerning the associated I/O device. The actual number of subchannels provided depends on the model and the configuration; the maximum number of subchannels is 65,536.

Channel Paths

I/O devices are attached through control units to the channel subsystem via channel paths. Control units may be attached to the channel subsystem via more than one channel path, and an I/O device may be attached to more than one control unit. In all, an individual I/O device may be accessible to a channel subsystem by as many as eight different channel paths, depending on the model and the configuration. The total number of channel paths provided by a channel subsystem depends on the model and the configuration; the maximum number of channel paths is 256.

A channel path can use one of three types of communication links:

- System/360 and System/370 I/O interface, called the parallel-I/O interface; the channel path is called a parallel channel path
- ESCON I/O interface, called a serial-I/O interface; the channel path is called a serial channel path
- FICON I/O interface, also called a serial-I/O interface; the channel path again is called a serial channel path

Each parallel-I/O interface consists of a number of electrical signal lines between the channel subsystem and one or more control units. Eight control units can share a single parallel-I/O interface. Up to 256 I/O devices can be addressed on a single parallel-I/O interface. The parallel-I/O interface is described in the publication IBM System/360 and System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers’ Information, GA22-6974.

Each serial-I/O interface consists of two optical-fiber conductors between any two of a channel subsystem, a dynamic switch, and a control unit. A dynamic switch can be connected by means of multiple serial-I/O interfaces to either the same or different channel subsystems and to multiple control units. The number of control units which
can be connected on one channel path depends on the channel-subsystem and dynamic-switch capabilities. Up to 256 devices can be attached to each control unit that uses the serial-I/O interface, depending on the control unit. The ESCON I/O interface is described in the publication *ESA/390 ESCON I/O Interface*, SA22-7202. The FICON I/O interface is described in the ANSI standards document *Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2)*.

**I/O Devices and Control Units**

I/O devices include such equipment as printers, magnetic-tape units, direct-access-storage devices, displays, keyboards, communications controllers, teleprocessing devices, and sensor-based equipment. Many I/O devices function with an external medium, such as paper or magnetic tape. Other I/O devices handle only electrical signals, such as those found in displays and communications networks. In all cases, I/O-device operation is regulated by a control unit that provides the logical and buffering capabilities necessary to operate the associated I/O device. From the programming point of view, most control-unit functions merge with I/O-device functions. The control-unit function may be housed with the I/O device or in the CPU, or a separate control unit may be used.

---

**Operator Facilities**

The operator facilities provide the functions necessary for operator control of the machine. Associated with the operator facilities may be an operator-console device, which may also be used as an I/O device for communicating with the program.

The main functions provided by the operator facilities include resetting, clearing, initial program loading, start, stop, alter, and display.
Chapter 3. Storage

This chapter discusses the representation of information in main storage, as well as addressing, protection, and reference and change recording. The aspects of addressing which are covered include the format of addresses, the concept of address spaces, the various types of addresses, and the manner in which one type of address is translated to another type of address. A list of
permanently assigned storage locations appears at the end of the chapter.

Main storage provides the system with directly addressable fast-access storage of data. Both data and programs must be loaded into main storage (from input devices) before they can be processed.

Main storage may include one or more smaller faster-access buffer storages, sometimes called caches. A cache is usually physically associated with a CPU or an I/O processor. The effects, except on performance, of the physical construction and use of distinct storage media are not observable by the program.

Fetching and storing of data by a CPU are not affected by any concurrent channel-subsystem activity or by a concurrent reference to the same storage location by another CPU. When concurrent requests to a main-storage location occur, access normally is granted in a sequence determined by the system. If a reference changes the contents of the location, any subsequent storage fetches obtain the new contents.

Main storage may be volatile or nonvolatile. If it is volatile, the contents of main storage are not preserved when power is turned off. If it is nonvolatile, turning power off and then back on does not affect the contents of main storage, provided all CPUs are in the stopped state and no references are made to main storage when power is being turned off. In both types of main storage, the contents of storage keys are not necessarily preserved when the power for main storage is turned off.

Note: Because most references in this publication apply to virtual storage, the abbreviated term “storage” is often used in place of “virtual storage.” The term “storage” may also be used in place of “main storage,” “absolute storage,” or “real storage” when the meaning is clear. The terms “main storage” and “absolute storage” are used to describe storage which is addressable by means of an absolute address. The terms describe fast-access storage, as opposed to auxiliary storage, such as that provided by direct-access storage devices. “Real storage” is synonymous with “absolute storage” except for the effects of pre-fixing.

Storage Addressing

Storage is viewed as a long horizontal string of bits. For most operations, accesses to storage proceed in a left-to-right sequence. The string of bits is subdivided into units of eight bits. An eight-bit unit is called a byte, which is the basic building block of all information formats.

Each byte location in storage is identified by a unique nonnegative integer, which is the address of that byte location or, simply, the byte address. Adjacent byte locations have consecutive addresses, starting with 0 on the left and proceeding in a left-to-right sequence. Addresses are unsigned binary integers and are 24, 31, or 64 bits. Addresses are described in “Address Size and Wraparound” on page 3-5.

Information Formats

Information is transmitted between storage and a CPU or the channel subsystem one byte, or a group of bytes, at a time. Unless otherwise specified, a group of bytes in storage is addressed by the leftmost byte of the group. The number of bytes in the group is either implied or explicitly specified by the operation to be performed. When used in a CPU operation, a group of bytes is called a field.

Within each group of bytes, bits are numbered in a left-to-right sequence. The leftmost bits are sometimes referred to as the “high-order” bits and the rightmost bits as the “low-order” bits. Bit numbers are not storage addresses, however. Only bytes can be addressed. To operate on individual bits of a byte in storage, it is necessary to access the entire byte.

The bits in a byte are numbered 0 through 7, from left to right.

The bits in an address may be numbered 8-31 or 40-63 for 24-bit addresses or 1-31 or 33-63 for 31-bit addresses; they are numbered 0-63 for 64-bit addresses. Within any other fixed-length format of multiple bytes, the bits making up the format are consecutively numbered starting from 0.

For purposes of error detection, and in some models for correction, one or more check bits may be transmitted with each byte or with a group of
bytes. Such check bits are generated automatically by the machine and cannot be directly controlled by the program. References in this publication to the length of data fields and registers exclude mention of the associated check bits. All storage capacities are expressed in number of bytes.

When the length of a storage-operand field is implied by the operation code of an instruction, the field is said to have a fixed length, which can be one, two, four, or eight bytes. Larger fields may be implied for some instructions.

When the length of a storage-operand field is not implied but is stated explicitly, the field is said to have a variable length. Variable-length operands can vary in length by increments of one byte.

When information is placed in storage, the contents of only those byte locations are replaced that are included in the designated field, even though the width of the physical path to storage may be greater than the length of the field being stored.

Integral Boundaries

Certain units of information must be on an integral boundary in storage. A boundary is called integral for a unit of information when its storage address is a multiple of the length of the unit in bytes. Special names are given to fields of 2, 4, 8, and 16 bytes on an integral boundary. A halfword is a group of two consecutive bytes on a two-byte boundary and is the basic building block of instructions. A word is a group of four consecutive bytes on a four-byte boundary. A doubleword is a group of eight consecutive bytes on an eight-byte boundary. A quadword is a group of 16 consecutive bytes on a 16-byte boundary. (See Figure 3-1 on page 3-4)

When storage addresses designate halfwords, words, doublewords, and quadwords, the binary representation of the address contains one, two, three, or four rightmost zero bits, respectively.

Instructions must be on two-byte integral boundaries, and CCWs, IDAWs, and the storage operands of certain instructions must be on other integral boundaries. The storage operands of most instructions do not have boundary-alignment requirements.

Programming Note: For fixed-field-length operations with field lengths that are a power of 2, significant performance degradation is possible when storage operands are not positioned at addresses that are integral multiples of the operand length. To improve performance, frequently used storage operands should be aligned on integral boundaries.

Address Types and Formats

Address Types

For purposes of addressing main storage, three basic types of addresses are recognized: absolute, real, and virtual. The addresses are distinguished on the basis of the transformations that are applied to the address during a storage access. Address translation converts virtual to real, and prefixing converts real to absolute. In addition to the three basic address types, additional types are defined which are treated as one or another of the three basic types, depending on the instruction and the current mode.

Absolute Address

An absolute address is the address assigned to a main-storage location. An absolute address is used for a storage access without any transformations performed on it.

The channel subsystem and all CPUs in the configuration refer to a shared main-storage location by using the same absolute address. Available main storage is usually assigned contiguous absolute addresses starting at 0, and the addresses are always assigned in complete 4K-byte blocks on integral boundaries. An exception is recognized when an attempt is made to use an absolute address in a block which has not been assigned to physical locations. On some models, storage-reconfiguration controls may be provided which permit the operator to change the correspondence between absolute addresses and physical locations. However, at any one time, a physical location is not associated with more than one absolute address.

Storage consisting of byte locations sequenced according to their absolute addresses is referred to as absolute storage.
Real Address
A real address identifies a location in real storage. When a real address is used for an access to main storage, it is converted, by means of prefixing, to an absolute address.

At any instant there is one real-address to absolute-address mapping for each CPU in the configuration. When a real address is used by a CPU to access main storage, it is converted to an absolute address by prefixing. The particular transformation is defined by the value in the prefix register for the CPU.

Storage consisting of byte locations sequenced according to their real addresses is referred to as real storage.

Virtual Address
A virtual address identifies a location in virtual storage. When a virtual address is used for an access to main storage, it is translated by means of dynamic address translation to a real address, which is then further converted by prefixing to an absolute address.

Primary Virtual Address
A primary virtual address is a virtual address which is to be translated by means of the primary address-space-control element. Logical addresses are treated as primary virtual addresses when in the primary-space mode. Instruction addresses are treated as primary virtual addresses when in the primary-space mode, secondary-space mode, or access-register mode. The first-operand address of MOVE TO PRIMARY and the second-operand address of MOVE TO SECONDARY are always treated as primary virtual addresses.

Secondary Virtual Address
A secondary virtual address is a virtual address which is to be translated by means of the secondary address-space-control element. Logical addresses are treated as secondary virtual addresses when in the secondary-space mode. The second-operand address of MOVE TO PRIMARY and the first-operand address of MOVE TO SECONDARY are always treated as secondary virtual addresses.
AR-Specified Virtual Address
An AR-specified virtual address is a virtual address which is to be translated by means of an access-register-specified address-space-control element. Logical addresses are treated as AR-specified addresses when in the access-register mode.

Home Virtual Address
A home virtual address is a virtual address which is to be translated by means of the home address-space-control element. Logical addresses and instruction addresses are treated as home virtual addresses when in the home-space mode.

Logical Address
Except where otherwise specified, the storage-operand addresses for most instructions are logical addresses. Logical addresses are treated as real addresses in the real mode, as primary virtual addresses in the primary-space mode, as secondary virtual addresses in the secondary-space mode, as AR-specified virtual addresses in the access-register mode, and as home virtual addresses in the home-space mode. Some instructions have storage-operand addresses or storage accesses associated with the instruction which do not follow the rules for logical addresses. In all such cases, the instruction definition contains a definition of the type of address.

Instruction Address
Addresses used to fetch instructions from storage are called instruction addresses. Instruction addresses are treated as real addresses in the real mode, as primary virtual addresses in the primary-space mode, secondary-space mode, or access-register mode, and as home virtual addresses in the home-space mode. The instruction address in the current PSW and the target address of EXECUTE are instruction addresses.

Effective Address
In some situations, it is convenient to use the term “effective address.” An effective address is the address which exists before any transformation by dynamic address translation or prefixing is performed. An effective address may be specified directly in a register or may result from address arithmetic. Address arithmetic is the addition of the base and displacement or of the base, index, and displacement.

Address Size and Wraparound
An address size refers to the maximum number of significant bits that can represent an address. Three sizes of addresses are provided: 24-bit, 31-bit, and 64-bit. A 24-bit address can accommodate a maximum of 16,777,216 (16M) bytes; with a 31-bit address, 2,147,483,648 (2G) bytes can be addressed; and, with a 64-bit address, 18,446,744,073,709,551,616 (16E) bytes can be addressed.

The bits of a 24-bit, 31-bit, or 64-bit address produced by address arithmetic under the control of the current addressing mode are numbered 40-63, 33-63, and 0-63, respectively, corresponding to the numbering of base-address and index bits in a general register:

- **24-Bit Address**
  - 0 40 63
- **31-Bit Address**
  - 0 33 63
- **64-Bit Address**
  - 0 63

The bits of an address that is 31 bits regardless of the addressing mode are numbered 1-31, and, when a 24-bit or 31-bit address is contained in a four-byte field in storage, the bits are numbered 8-31 or 1-31, respectively:

- **24-Bit Address**
  - 0 8 31
- **31-Bit Address**
  - 0 1 31

A 24-bit or 31-bit virtual address is expanded to 64 bits by appending 40 or 33 zeros, respectively, on the left before it is translated by means of the DAT process, and a 24-bit or 31-bit real address is similarly expanded to 64 bits before it is transformed by prefixing. A 24-bit or 31-bit absolute address is expanded to 64 bits before main storage is accessed. Thus, the 24-bit address always designates a location in the first 16M-byte range of main storage.
block of the 16E-byte storage addressable by a 64-bit address, and the 31-bit address always designates a location in the first 2G-byte block.

Unless specifically stated to the contrary, the following definition applies in this publication: whenever the machine generates and provides to the program a 24-bit or 31-bit address, the address is made available (placed in storage or loaded into a general register) by being imbedded in a 32-bit field, with the leftmost eight bits or one bit in the field, respectively, set to zeros. When the address is loaded into a general register, bits 0-31 of the register remain unchanged.

The size of effective addresses is controlled by bits 31 and 32 of the PSW, the extended-addressing-mode bit and the basic-addressing-mode bit, respectively. When bits 31 and 32 are both zero, the CPU is in the 24-bit addressing mode, and 24-bit operand and instruction effective addresses are specified. When bit 31 is zero and bit 32 is one, the CPU is in the 31-bit addressing mode, and 31-bit operand and instruction effective addresses are specified. When bits 31 and 32 are both one, the CPU is in the 64-bit addressing mode, and 64-bit operand and instruction effective addresses are specified (see "Address Generation" on page 5-8).

The sizes of the real or absolute addresses used or yielded by the ASN-translation, ASN-authorization, PC-number-translation, and access-register-translation processes are always 31 bits regardless of the current addressing mode. Similarly, the sizes of the real or absolute addresses used or yielded by the DAT, stacking, unstacking, and tracing processes are always 64 bits.

The size of the data address in a CCW is under control of the CCW-format-control bit in the operation-request block (ORB) designated by a START SUBCHANNEL instruction. The CCWs with 24-bit and 31-bit addresses are called format-0 and format-1 CCWs, respectively. Format-0 and format-1 CCWs are described in Chapter 15, "Basic I/O Functions." Similarly, the size of the data address in an IDAW is under control of the IDAW-format-control bit in the ORB. The IDAWs with 31-bit and 64-bit addresses are called format-1 and format-2 IDAWs, respectively. Format-1 and format-2 IDAWs are described in Chapter 15, “Basic I/O Functions.”

Address Wraparound
The CPU performs address generation when it forms an operand or instruction address or when it generates the address of a table entry from the appropriate table origin and index. It also performs address generation when it increments an address to access successive bytes of a field. Similarly, the channel subsystem performs address generation when it increments an address (1) to fetch a CCW, (2) to fetch an IDAW, (3) to transfer data, or (4) to compute the address of an I/O measurement block.

When, during the generation of the address, an address is obtained that exceeds the value allowed for the address size ($2^{34} - 1$, $2^{31} - 1$, or $2^{64} - 1$), one of the following two actions is taken:

1. The carry out of the high-order bit position of the address is ignored. This handling of an address of excessive size is called wraparound.
2. An interruption condition is recognized.

The effect of wraparound is to make an address space appear circular; that is, address 0 appears to follow the maximum allowable address. Address arithmetic and wraparound occur before transformation, if any, of the address by DAT or prefixing.

Addresses generated by the CPU that may be virtual addresses always wrap. Wraparound also occurs when the linkage-stack-entry address in control register 15 is decremented below 0 by PROGRAM RETURN. For CPU table entries that are addressed by real or absolute addresses, it is unpredictable whether the address wraps or an addressing exception is recognized.

For channel-program execution, when the generated address exceeds the value for the address size (or, for the read-backward command is decremented below 0), an I/O program-check condition is recognized.

Figure 3-2 on page 3-7 identifies what limit values apply to the generation of different addresses and how addresses are handled when they exceed the allowed value.
<table>
<thead>
<tr>
<th>Address Generation for</th>
<th>Address Type</th>
<th>Handling when Address Would Wrap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions and operands when EAM and BAM are zero</td>
<td>L,I,R,V</td>
<td>W24</td>
</tr>
<tr>
<td>Successive bytes of instructions and operands when EAM and BAM are zero</td>
<td>I,L,V&lt;sup&gt;1&lt;/sup&gt;</td>
<td>W24</td>
</tr>
<tr>
<td>Instructions and operands when EAM is zero and BAM is one</td>
<td>L,I,R,V</td>
<td>W31</td>
</tr>
<tr>
<td>Successive bytes of instructions and operands when EAM is zero and BAM is one</td>
<td>I,L,V&lt;sup&gt;1&lt;/sup&gt;</td>
<td>W31</td>
</tr>
<tr>
<td>Instructions and operands when EAM and BAM are one</td>
<td>L,I,R,V</td>
<td>W64</td>
</tr>
<tr>
<td>Successive bytes of instructions and operands when EAM and BAM are one</td>
<td>I,L,V&lt;sup&gt;1&lt;/sup&gt;</td>
<td>W64</td>
</tr>
<tr>
<td>DAT-table entries when used for implicit translation or LRA or STRAG</td>
<td>A or R&lt;sup&gt;2&lt;/sup&gt;</td>
<td>X64</td>
</tr>
<tr>
<td>ASN-second-table, authority-table (during ASN authorization), linkage-table, linkage-first-table, linkage-second-table, and entry-table entries</td>
<td>R</td>
<td>X31</td>
</tr>
<tr>
<td>Authority-table (during access-register translation) and access-list entries</td>
<td>A or R&lt;sup&gt;2&lt;/sup&gt;</td>
<td>X31</td>
</tr>
<tr>
<td>Linkage-stack entry</td>
<td>V</td>
<td>W64</td>
</tr>
<tr>
<td>I/O measurement block</td>
<td>A</td>
<td>P31</td>
</tr>
<tr>
<td>For a channel program with format-0 CCWs:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successive CCWs</td>
<td>A</td>
<td>P24</td>
</tr>
<tr>
<td>Successive IDAWs</td>
<td>A</td>
<td>P24</td>
</tr>
<tr>
<td>Successive bytes of I/O data (without IDAWs)</td>
<td>A</td>
<td>P24</td>
</tr>
<tr>
<td>Successive bytes of I/O data (with format-1 IDAWs)</td>
<td>A</td>
<td>P31</td>
</tr>
<tr>
<td>Successive bytes of I/O data (with format-2 IDAWs)</td>
<td>A</td>
<td>P64</td>
</tr>
<tr>
<td>For a channel program with format-1 CCWs:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successive CCWs</td>
<td>A</td>
<td>P31</td>
</tr>
<tr>
<td>Successive IDAWs</td>
<td>A</td>
<td>P31</td>
</tr>
<tr>
<td>Successive bytes of I/O data (without IDAWs)</td>
<td>A</td>
<td>P31</td>
</tr>
<tr>
<td>Successive bytes of I/O data (with format-1 IDAWs)</td>
<td>A</td>
<td>P31</td>
</tr>
<tr>
<td>Successive bytes of I/O data (with format-2 IDAWs)</td>
<td>A</td>
<td>P64</td>
</tr>
</tbody>
</table>

Figure 3-2 (Part 1 of 2). Address Wraparound
Explanation:

1. Real addresses do not apply in this case since the instructions which designate operands by means of real addresses cannot designate operands that cross boundary 2^{24}, 2^{31}, or 2^{64}.
2. It is unpredictable whether the address is absolute or real.

A Absolute address.
BAM Basic-addressing-mode bit in the PSW.
EAM Extended-addressing-mode bit in the PSW.
I Instruction address.
L Logical address.
P24 An I/O program-check condition is recognized when the address exceeds 2^{24} - 1 or is decremented below zero.
P31 An I/O program-check condition is recognized when the address exceeds 2^{31} - 1 or is decremented below zero.
P64 An I/O program-check condition is recognized when the address exceeds 2^{64} - 1 or is decremented below zero.
R Real address.
V Virtual address.
W24 Wrap to location 0 after location 2^{24} - 1 and vice versa.
W31 Wrap to location 0 after location 2^{31} - 1 and vice versa.
W64 Wrap to location 0 after location 2^{64} - 1 and vice versa.
X31 When the address exceeds 2^{31} - 1, it is unpredictable whether the address wraps to location 0 after location 2^{31} - 1 or whether an addressing exception is recognized.
X64 When the address exceeds 2^{64} - 1, it is unpredictable whether the address wraps to location 0 after location 2^{64} - 1 or whether an addressing exception is recognized.

Figure 3-2 (Part 2 of 2). Address Wraparound

Storage Key

A storage key is associated with each 4K-byte block of storage that is available in the configuration. The storage key has the following format:

```
ACC | F | R | C
```

| 0 | 4 | 6 |

The bit positions in the storage key are allocated as follows:

**Access-Control Bits (ACC):** If a reference is subject to key-controlled protection, the four access-control bits, bits 0-3, are matched with the four-bit access key when information is stored and when information is fetched from a location that is protected against fetching.

**Fetch-Protection Bit (F):** If a reference is subject to key-controlled protection, the fetch-protection bit, bit 4, controls whether key-controlled protection applies to fetch-type references: a zero indicates that only store-type references are monitored and that fetching with any access key is permitted; a one indicates that key-controlled protection applies to both fetching and storing. No distinction is made between the fetching of instructions and of operands.

**Reference Bit (R):** The reference bit, bit 5, normally is set to one each time a location in the corresponding storage block is referred to either for storing or for fetching of information.

**Change Bit (C):** The change bit, bit 6, is set to one each time information is stored at a location in the corresponding storage block.

Storage keys are not part of addressable storage. The entire storage key is set by SET STORAGE KEY EXTENDED and inspected by INSERT STORAGE KEY EXTENDED. Additionally, the instruction RESET REFERENCE BIT EXTENDED provides a means of inspecting the reference and change bits and of setting the reference bit to zero. Bits 0-4 of the storage key are inspected by the INSERT VIRTUAL STORAGE KEY instruction. The contents of the storage key are unpredictable during and after the execution of the usability test of the TEST BLOCK instruction.
Protection

Four protection facilities are provided to protect the contents of main storage from destruction or misuse by programs that contain errors or are unauthorized: key-controlled protection, access-list-controlled protection, page protection, and low-address protection. The protection facilities are applied independently; access to main storage is only permitted when none of the facilities prohibits the access.

Key-controlled protection affords protection against improper storing or against both improper storing and fetching, but not against improper fetching alone.

Key-Controlled Protection

When key-controlled protection applies to a storage access, a store is permitted only when the storage key matches the access key associated with the request for storage access; a fetch is permitted when the keys match or when the fetch-protection bit of the storage key is zero.

The keys are said to match when the four access-control bits of the storage key are equal to the access key, or when the access key is zero.

The protection action is summarized in Figure 3-3.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Is Access to Storage Permitted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch-Protection Bit of</td>
<td></td>
</tr>
<tr>
<td>Storage Key</td>
<td>Key Relation</td>
</tr>
<tr>
<td>0</td>
<td>Match</td>
</tr>
<tr>
<td>0</td>
<td>Mismatch</td>
</tr>
<tr>
<td>1</td>
<td>Match</td>
</tr>
<tr>
<td>1</td>
<td>Mismatch</td>
</tr>
</tbody>
</table>

Explanations:
- Match: The four access-control bits of the storage key are equal to the access key, or the access key is zero.
- Yes: Access is permitted.
- No: Access is not permitted. On fetching, the information is not made available to the program; on storing, the contents of the storage location are not changed.

![Figure 3-3. Summary of Protection Action](image)

When a CPU access is prohibited because of key-controlled protection, the execution of the instruction is terminated, and a program interruption for a protection exception takes place. However, the unit of operation or the execution of the instruction may be suppressed, as described in the section "Suppression on Protection" on page 3-12. When a channel-program access is prohibited, the start function is ended, and the protection-check condition is indicated in the associated interruption-response block (IRB). When a measurement-block access is prohibited, the I/O measurement-block protection-check condition is indicated.

When a store access is prohibited because of key-controlled protection, the contents of the protected location remain unchanged. When a fetch access is prohibited, the protected information is not loaded into a register, moved to another storage location, or provided to an I/O device. For a prohibited instruction fetch, the instruction is suppressed, and an arbitrary instruction-length code is indicated.

Key-controlled protection is independent of whether the CPU is in the problem or the supervisor state and, except as described below, does not depend on the type of CPU instruction or channel-command word being executed.
Except where otherwise specified, all accesses to storage locations that are explicitly designated by the program and that are used by the CPU to store or fetch information are subject to key-controlled protection.

Key-controlled protection does not apply when the storage-protection-override control is one and the value of the four access-control bits of the storage key is 9. Key-controlled protection for fetches may or may not apply when the fetch-protection-override control is one, depending on the effective address and the private-space control.

The storage-protection-override control and fetch-protection-override control do not affect storage references made by the channel subsystem.

Accesses to the second operand of TEST BLOCK are not subject to key-controlled protection.

All storage accesses by the channel subsystem to access the I/O measurement block, or by a channel program to fetch a CCW or IDAW or to access a data area designated during the execution of a CCW, are subject to key-controlled protection. However, if a CCW, an IDAW, or output data is prefetched, a protection check is not indicated until the CCW or IDAW is due to take control or until the data is due to be written.

Key-controlled protection is not applied to accesses that are implicitly made for any of such sequences as:

- An interruption
- CPU logout
- Fetching of table entries for access-register translation, dynamic-address translation, PC-number translation, ASN translation, or ASN authorization
- Tracing
- A store-status function
- Storing in real locations 184-191 when TEST PENDING INTERRUPTION has an operand address of zero
- Initial program loading

Similarly, protection does not apply to accesses initiated via the operator facilities for altering or displaying information. However, when the program explicitly designates these locations, they are subject to protection.

### Storage-Protection-Override Control

Bit 39 of control register 0 is the storage-protection-override control. When this bit is one, storage-protection override is active. When this bit is zero, storage-protection override is inactive. When storage-protection override is active, key-controlled storage protection is ignored for storage locations having an associated storage-key value of 9. When storage-protection override is inactive, no special action is taken for a storage-key value of 9.

Storage-protection override applies to instruction fetch and to the fetch and store accesses of instructions whose operand addresses are logical, virtual, or real. It does not apply to accesses made for the purpose of channel-program execution or for the purpose of channel-subsystem monitoring.

Storage-protection override has no effect on accesses which are not subject to key-controlled protection.

### Programming Notes:

1. Storage-protection override can be used to improve reliability in the case when a possibly erroneous application program is executed in conjunction with a reliable subsystem, provided that the application program needs to access only a portion of the storage accessed by the subsystem. The technique for doing this is as follows. The storage accessed by the application program is given storage key 9. The storage accessed by only the subsystem is given some other nonzero storage key, for example, key 8. The application is executed with PSW key 9. The subsystem is executed with PSW key 8 (in this example). As a result, the subsystem can access both the key-8 and the key-9 storage, while the application program can access only the key-9 storage.

2. Storage-protection override affects the accesses to storage made by the CPU and also affects the result set by TEST PROTECTION. However, those instructions which, in the problem state, test the PSW-key mask to determine if a particular key value may be used are not affected by whether storage-protection override is active. These instructions include, among others, MOVE WITH KEY and SET PSW KEY FROM...
ADDRESS. To permit these instructions to use an access key of 9 in the problem state, bit 9 of the PSW-key mask must be one.

**Fetch-Protection-Override Control**

Bit 38 of control register 0 is the fetch-protection-override control. When the bit is one, fetch protection is ignored for locations at effective addresses 0-2047. An effective address is the address which exists before any transformation by dynamic address translation or prefixing. However, fetch protection is not ignored if the effective address is subject to dynamic address translation and the private-space control, bit 55, is one in the address-space-control element used in the translation.

Fetch-protection override applies to instruction fetch and to the fetch accesses of instructions whose operand addresses are logical, virtual, or real. It does not apply to fetch accesses made for the purpose of channel-program execution or for the purpose of channel-subsystem monitoring. When this bit is set to zero, fetch protection of locations at effective addresses 0-2047 is determined by the state of the fetch-protection bit of the storage key associated with those locations.

Fetch-protection override has no effect on accesses which are not subject to key-controlled protection.

**Programming Note:** The fetch-protection-override control allows fetch protection of locations at addresses 2048-4095 along with no fetch protection of locations at addresses 0-2047.

**Access-List-Controlled Protection**

In the access-register mode, bit 6 of the access-list entry, the fetch-only bit, controls which types of operand references are permitted to the address space specified by the access-list entry. When the entry is used in the access-register-translation part of a reference and bit 6 is zero, both fetch-type and store-type references are permitted; when bit 6 is one, only fetch-type references are permitted, and an attempt to store causes a protection exception to be recognized and the execution of the instruction to be suppressed.

The fetch-only bit is included in the ALB access-list entry. A change to the fetch-only bit in an access-list entry in main storage does not necessarily have an immediate, if any, effect on whether a protection exception is recognized. However, this change to the bit will have an effect immediately after PURGE ALB or a COMPARE AND SWAP AND PURGE instruction that purges the ALB is executed.

TEST PROTECTION takes into consideration access-list-controlled protection when the CPU is in the access-register mode. A violation of access-list-controlled protection causes condition code 1 to be set, except that it does not prevent condition code 2 or 3 from being set when the conditions for those codes are satisfied.

**Programming Note:** A violation of access-list-controlled protection always causes suppression. A violation of any of the other protection types may cause termination.

**Page Protection**

The page-protection facility controls access to virtual storage by using the page-protection bit in each page-table entry and segment-table entry. It provides protection against improper storing.

The page-protection bit, bit 54, of the page-table entry controls whether storing is allowed into the corresponding 4K-byte page. When the bit is zero, both fetching and storing are permitted; when the bit is one, only fetching is permitted. When an attempt is made to store into a protected page, the contents of the page remain unchanged, the unit of operation or the execution of the instruction is suppressed, and a program interruption for protection takes place.

The page-protection bit, bit 54, of the segment-table entry is treated as being ORed into the page-protection-bit position of each entry in the page table designated by the segment-table entry. Thus, when the segment-table-entry page-protection bit is one, the effect is as if the page-protection bit were one in each entry in the designated page table.

Page protection applies to all store-type references that use a virtual address.
Low-Address Protection

The low-address-protection facility provides protection against the destruction of main-storage information used by the CPU during interruption processing. This is accomplished by prohibiting instructions from storing with effective addresses in the ranges 0 through 511 and 4096 through 4607 (the first 512 bytes of each of the first and second 4K-byte effective-address blocks). The range criterion is applied before address transformation, if any, of the address by dynamic address translation or prefixing. However, the range criterion is not applied, with the result that low-address protection does not apply, if the effective address is subject to dynamic address translation and the private-space control, bit 55, is one in the address-space-control element used in the translation. Low-address protection does not apply if the address-space-control element to be used is not available due to another type of exception.

Low-address protection is under control of bit 35 of control register 0, the low-address-protection-control bit. When the bit is zero, low-address protection is off; when the bit is one, low-address protection is on.

If an access is prohibited because of low-address protection, the contents of the protected location remain unchanged, the execution of the instruction is terminated, and a program interruption for a protection exception takes place. However, the unit of operation or the execution of the instruction may be suppressed, as described in the section “Suppression on Protection.”

Any attempt by the program to store by using effective addresses in the range 0 through 511 or 4096 through 4607 is subject to low-address protection. Low-address protection is applied to the store accesses of instructions whose operand addresses are logical, virtual, or real. Low-address protection is also applied to the trace table.

Low-address protection is not applied to accesses made by the CPU or the channel subsystem for such sequences as interruptions, CPU logout, the storing of the I/O-interruption code in real locations 184-191 by TEST PENDING INTERRUPTION, and the initial-program-loading and store-status functions, nor is it applied to data stores during I/O data transfer. However, explicit stores by a program at any of these locations are subject to low-address protection.

Programming Notes:

1. Low-address protection and key-controlled protection apply to the same store accesses, except that:

   a. Low-address protection does not apply to storing performed by the channel subsystem, whereas key-controlled protection does.

   b. Key-controlled protection does not apply to tracing, the second operand of TEST BLOCK, or instructions that operate specifically on the linkage stack, whereas low-address protection does.

2. Because fetch-protection override and low-address protection do not apply to an address space for which the private-space control is one in the address-space-control element, locations 0-2047 and 4096-4607 in the address space are usable the same as the other locations in the space.

Suppression on Protection

Some instruction definitions specify that the operation is always suppressed if a protection exception due to any type of protection is recognized. When that specification is absent, the execution of an instruction is always suppressed if a protection exception due to access-list-controlled protection or page protection is recognized, and it may be either suppressed or terminated if a protection exception due to low-address protection or key-controlled protection is recognized.

The suppression-on-protection function allows the control program to locate the segment-table entry and page-table entry used in the translation of a virtual address that caused a protection exception, in order to determine if the exception was due to page protection.¹ This is necessary for the imple-

¹ The suppression-on-protection function originated as the ESA/390 suppression-on-protection facility. Suppression for page protection was new as part of that facility.
mentation of the Posix fork function (discussed in a programming note). The function also allows the control program to avoid locating the segment-table and page-table entries if the address was not virtual or the exception was due to access-list-controlled protection.

During a program interruption due to a protection exception, either a one or a zero is stored in bit position 61 of real locations 168-175. The storing of a one in bit position 61 indicates that:

- The unit of operation or instruction execution during which the exception was recognized was suppressed.
- If dynamic address translation (DAT) was on, as indicated by the DAT-mode bit in the program old PSW, the effective address that caused the exception is one that was to be translated by DAT. (The effective address is the address which exists before any transformation by DAT or prefixing.) Bit 61 is set to zero if DAT was on but the effective address was not to be translated by DAT because it is a real address. If DAT was off, the protection exception cannot have been due to page protection.

- Bit positions 0-51 of real locations 168-175 contain bits 0-51 of the effective address that caused the exception. If DAT was on, indicating that the effective address was to be translated by DAT, bit positions 62 and 63 of real locations 168-175, and real location 160, contain the same information as is stored during a program interruption due to a page-translation exception — this information identifies the address space containing the protected address. Also, bit 60 of real locations 168-175 is zero if the protection exception was not due to access-list-controlled protection or is one if the exception was due to access-list-controlled protection. A one in bit position 60 indicates that the exception was not due to page protection. If DAT was off, the contents of bit positions 60, 62, and 63 of real locations 168-175, and the contents of real location 160, are unpredictable. The contents of bit positions 52-59 of real locations 168-175 are always unpredictable.

Bit 61 being zero indicates that the operation was either suppressed or terminated and that the contents of the remainder of real locations 168-175, and of real location 160 are unpredictable.

Bit 61 is set to one if the protection exception was due to access-list-controlled protection or page protection. Bit 61 may be set to one if the protection exception was due to low-address protection or key-controlled protection.

If a protection-exception condition exists due to either access-list-controlled protection or page protection but also exists due to either low-address protection or key-controlled protection, it is unpredictable whether bit 61 is set to zero or one.

**Programming Notes:**

1. The suppression-on-protection function is useful in performing the Posix fork function, which causes a duplicate address space to be created. When forking occurs, the control program causes the same page of different address spaces to map to a single page frame of real storage so long as a store in the page is not attempted. Then, when a store is attempted in a particular address space, the control program assigns a unique page frame to the page in that address space and copies the contents of the page to the new page frame. This last action is sometimes called the copy-on-write function. The control program may initially set the page-protection bit to one in the page-table entry for a page in order to detect an attempt to store in the page. The control program may initially set the page-protection bit to one in a segment-table entry to detect an attempt to store anywhere in the specified segment.

2. Bit 61 being one in real locations 168-175 when DAT was on indicates that the address that caused a protection exception is virtual. This indication allows programmed forms of access-register translation and dynamic address translation to be performed to determine whether the exception was due to page protection as opposed to low-address or key-controlled protection.

3. The results of suppression on protection are summarized in Figure 3-4 on page 3-14.
Reference recording provides information for use in selecting pages for replacement. Reference recording uses the reference bit, bit 5 of the storage key. The reference bit is set to one each time a location in the corresponding storage block is referred to either for fetching or for storing information, regardless of whether DAT is on or off.

Reference recording is always active and takes place for all storage accesses, including those made by any CPU, any operator facility, or the channel subsystem. It takes place for implicit accesses made by the machine, such as those which are part of interruptions and I/O-instruction execution.

Reference recording does not occur for operand accesses of the following instructions since they directly refer to a storage key without accessing a storage location:

- INSERT STORAGE KEY EXTENDED
- RESET REFERENCE BIT EXTENDED (reference bit is set to zero)
- SET STORAGE KEY EXTENDED (reference bit is set to a specified value)

The record provided by the reference bit is substantially accurate. The reference bit may be set to one by fetching data or instructions that are neither designated nor used by the program, and, under certain conditions, a reference may be made without the reference bit being set to one. Under certain unusual circumstances, a reference bit may be set to zero by other than explicit program action.

Change Recording

Change recording provides information as to which pages have to be saved in auxiliary storage when they are replaced in main storage. Change recording uses the change bit, bit 6 of the storage key.

The change bit is set to one each time a store access causes the contents of the corresponding storage block to be changed. A store access that does not change the contents of storage may or may not set the change bit to one.

The change bit is not set to one for an attempt to store if the access is prohibited. In particular:

1. For the CPU, a store access is prohibited whenever an access exception exists for that access, or whenever an exception exists which is of higher priority than the priority of an access exception for that access.

2. For the channel subsystem, a store access is prohibited whenever a key-controlled-protection violation exists for that access.

Change recording is always active and takes place for all store accesses to storage, including those made by any CPU, any operator facility, or the channel subsystem. It takes place for implicit references made by the machine, such as those which are part of interruptions.
Change recording does not take place for the operands of the following instructions since they directly modify a storage key without modifying a storage location:

- RESET REFERENCE BIT EXTENDED
- SET STORAGE KEY EXTENDED (change bit is set to a specified value)

Change bits which have been changed from zeros to ones are not necessarily restored to zeros on CPU retry (see "CPU Retry" on page 11-2). See "Exceptions to Nullification and Suppression" on page 5-23 for a description of the handling of the change bit in certain unusual situations.

### Prefixing

Prefixing provides the ability to assign the range of real addresses 0-8191 to a different block in absolute storage for each CPU, thus permitting more than one CPU sharing main storage to operate concurrently with a minimum of interference, especially in the processing of interruptions.

Prefixing causes real addresses in the range 0-8191 to correspond one-for-one to the block of 8K-byte absolute addresses (the prefix area) identified by the value in bit positions 0-50 of the prefix register for the CPU, and the block of real addresses identified by that value in the prefix register to correspond one-for-one to absolute addresses 0-8191. The remaining real addresses are the same as the corresponding absolute addresses. This transformation allows each CPU to access all of main storage, including the first 8K bytes and the locations designated by the prefix registers of other CPUs.

The relationship between real and absolute addresses is graphically depicted in Figure 3-5 on page 3-16.

The prefix is a 51-bit quantity contained in bit positions 0-50 of the prefix register. The register has the following format:

```
0000 Prefix Bits 33-50

0 33 51 63
```

Bits 0-32 of the register are always all zeros. Bits 33-50 of the register can be set and inspected by the privileged instructions SET PREFIX and STORE PREFIX, respectively.

SET PREFIX sets bits 33-50 of the prefix register with the value in bit positions 1-18 of a word in storage, and it ignores the contents of bit positions 0 and 19-31 of the word. STORE PREFIX stores the value in bit positions 33-50 of the prefix register in bit positions 1-18 of a word in storage, and it stores zeros in bit positions 0 and 19-31 of the word.

When the contents of the prefix register are changed, the change is effective for the next sequential instruction.

When prefixing is applied, the real address is transformed into an absolute address by using one of the following rules, depending on bits 0-50 of the real address:

1. Bits 0-50 of the address, if all zeros, are replaced with bits 0-50 of the prefix.
2. Bits 0-50 of the address, if equal to bits 0-50 of the prefix, are replaced with zeros.
3. Bits 0-50 of the address, if not all zeros and not equal to bits 0-50 of the prefix, remain unchanged.

Only the address presented to storage is translated by prefixing. The contents of the source of the address remain unchanged.

The distinction between real and absolute addresses is made even when the prefix register contains all zeros, in which case a real address and its corresponding absolute address are identical.
(1) Real addresses in which bits 0-50 are equal to bits 0-50 of the prefix for this CPU (A or B).

(2) Absolute addresses of the block that contains for this CPU (A or B) the real locations 0-8191.

Figure 3-5. Relationship between Real and Absolute Addresses

Address Spaces

An address space is a consecutive sequence of integer numbers (virtual addresses), together with the specific transformation parameters which allow each number to be associated with a byte location in storage. The sequence starts at zero and proceeds left to right.

When a virtual address is used by a CPU to access main storage, it is first converted, by means of dynamic address translation (DAT), to a real address, and then, by means of prefixing, to an absolute address. DAT may use from five to two levels of tables (region first table, region second table, region third table, segment table, and page table) as transformation parameters. The designation (origin and length) of the highest-level table for a specific address space is called an address-space-control element, and it is found for use by DAT in a control register or as specified by an access register. Alternatively, the address-space-control element for an address space may be a real-space designation, which indicates that DAT is to translate the virtual address simply by treating it as a real address and without using any tables.

DAT uses, at different times, the address-space-control elements in different control registers or specified by the access registers. The choice is determined by the translation mode specified in the current PSW. Four translation modes are available: primary-space mode, secondary-space mode, access-register mode, and home-space mode. Different address spaces are addressable depending on the translation mode.

At any instant when the CPU is in the primary-space mode or secondary-space mode, the CPU can translate virtual addresses belonging to two address spaces — the primary address space and the secondary address space. At any instant when the CPU is in the access-register mode, it can translate virtual addresses of up to 16 address spaces — the primary address space and up to 15 AR-specified address spaces. At any instant when the CPU is in the home-space
mode, it can translate virtual addresses of the home address space.

The primary address space is identified as such because it consists of primary virtual addresses, which are translated by means of the primary address-space-control element (ASCE). Similarly, the secondary address space consists of secondary virtual addresses translated by means of the secondary ASCE, the AR-specified address spaces consist of AR-specified virtual addresses translated by means of AR-specified ASCEs, and the home address space consists of home virtual addresses translated by means of the home ASCE. The primary and secondary ASCEs are in control registers 1 and 7, respectively. The AR-specified ASCEs are in control registers 1 and 7 and in table entries called ASN-second-table entries. The home ASCE is in control register 13.

Changing to Different Address Spaces

A program can cause different address spaces to be addressable by using the semiprivileged SET ADDRESS SPACE CONTROL or SET ADDRESS SPACE CONTROL FAST instruction to change the translation mode to the primary-space mode, secondary-space mode, access-register mode, or home-space mode. However, SET ADDRESS SPACE CONTROL and SET ADDRESS SPACE CONTROL FAST can set the home-space mode only in the supervisor state. The program can cause still other address spaces to be addressable by using unprivileged instructions to change the contents of the access registers and by using semiprivileged or privileged instructions to change the address-space-control elements in control registers 1 and 7. The semiprivileged instructions are ones that cause linkage from one address space to another and are the BRANCH IN SUBSPACE GROUP, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE instructions. The privileged instructions are the LOAD ADDRESS SPACE PARAMETERS and LOAD CONTROL instructions. Only LOAD CONTROL is available for changing the home address-space-control element in control register 13.

Address-Space Number

An address space may be assigned an address-space number (ASN) by the control program. The ASN designates, within a two-level table structure in main storage, an ASN-second-table entry containing information about the address space. If the ASN-second-table entry is marked as valid, it contains the address-space-control element that defines the address space.

Under certain circumstances, the semiprivileged instructions which place a new address-space-control element in control register 1 or 7 fetch this element from an ASN-second-table entry. Some of these instructions use an ASN-translation mechanism which, given an ASN, can locate the designated ASN-second-table entry.

The 16-bit unsigned binary format of the ASN permits 64K unique ASNs.

The ASNs for the primary and secondary address spaces are assigned positions in control registers. The ASN for the primary address space, called the primary ASN, is assigned bits 48-63 in control register 4, and that for the secondary address space, called the secondary ASN, is assigned bits 48-63 in control register 3. Bits 48-63 of these registers have the following formats:

Control Register 4

<table>
<thead>
<tr>
<th>PASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

Control Register 3

<table>
<thead>
<tr>
<th>SASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

A semiprivileged instruction that loads the primary or secondary address-space-control element into the appropriate control register also loads the corresponding ASN into the appropriate control register.

The ASN for the home address space is not assigned a position in a control register.

An access register containing the value 0 or 1 specifies the primary or secondary address space, respectively; and the address-space-control
element specified by the access register is in control register 1 or 7, respectively. An access register containing any other value designates an entry in a table called an access list. The designated access-list entry contains the real address of an ASN-second-table entry for the address space specified by the access register. The address-space-control element specified by the access register is in the ASN-second-table entry. Translating the contents of an access register to obtain an address-space-control element for use by DAT does not involve the use of an ASN.

**Note:** Virtual storage consisting of byte locations ordered according to their virtual addresses in an address space is usually referred to as “storage.”

**Programming Note:** Because an ASN-second-table entry is located from an access-list entry by means of its address instead of by means of its ASN, the ASN-second-table entries designated by access-list entries can be “pseudo” ASN-second-table entries, that is, entries which are not in the two-level structure able to be indexed by means of the ASN-translation process. The number of unique pseudo ASN-second-table entries can be greater than the number of unique ASNs and is limited only by the amount of storage available to be occupied by the ASN-second-table entries. Thus, in a sense, there is no limit on the number of possible address spaces.

### ASN-Second-Table-Entry Sequence Number

The ASN-second-table entry contains an ASN-second-table-entry sequence number (ASTESN) that may be used to control storage references to the related address space by means of an access register or through use of the SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, or LOAD ADDRESS SPACE PARAMETERS instruction and that may be used to control linkages to or back to the address space by means of the BRANCH IN SUBSPACE GROUP, PROGRAM CALL, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, and PROGRAM RETURN instructions. These uses of the ASTESN are described below.

When any instruction uses an access register to designate an access-list entry, which in turn designates an ASN-second-table entry, to perform a storage reference to the address space specified by the ASN-second-table entry, an ASTESN in the access-list entry is compared to the ASTESN in the ASN-second-table entry, and these ASTESNs must be equal; otherwise, an ASTE-sequence exception is recognized. This use of the ASTESN allows an access-list entry to be made usable if some authorization policy is changed or the designated ASN-second-table entry is reassigned to specify a conceptually different address space. The entry can be made usable by changing the ASTESN in the ASN-second-table entry. The use is further described in “Revoking Accessing Capability” on page 5-49.

The ASTESN is used in connection with subspace groups as follows:

- When BRANCH IN SUBSPACE GROUP uses an access register to designate an access-list entry, which in turn designates an ASN-second-table entry, to transfer control to the subspace specified by the ASN-second-table entry, an ASTESN in the access-list entry is compared to the ASTESN in the ASN-second-table entry, and these ASTESNs must be equal.

- When BRANCH IN SUBSPACE GROUP uses, in an access register, an access-list-entry token with the value 00000001 hex to transfer control to the subspace specified by the subspace-ASN-second-table-entry origin in the current dispatchable-unit control table (designated by control register 2), a subspace-ASN-second-table-entry sequence number (SSASTESN) in the dispatchable-unit control table is compared to the ASTESN in the ASN-second-table entry for the subspace, and the SSASTESN and ASTESN must be equal.

- When LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE sets the primary address-space-control element in control register 1 or the secondary address-space-control element in control register 7 with bits of the address-space-control element in the ASN-second-table entry for a subspace, the SSASTESN is compared to the ASTESN in the ASN-second-table entry for the sub-
space, and the SSASTESN and ASTESN must be equal.

Otherwise, in all of the above three cases, an ASTE-sequence exception is recognized, except that LOAD ADDRESS SPACE PARAMETERS sets condition code 1 or 2 depending on whether the subspace is the primary address space or the secondary address space, respectively. These uses of the ASTESN allow an access-list entry and the subspace-ASN-second-table-entry origin in the current dispatchable-unit control table to be made unusable when the ASN-second-table entry either of them designates is reassigned to specify a conceptually different address space. The access-list entry and subspace-ASN-second-table-entry origin can be made unusable by changing the ASTESN in the ASN-second-table entry. The uses are further described in "Subspace Groups" on page 5-64.

Programming Note: The above operations use an ASN-second-table-entry origin in either an access-list entry or the dispatchable-unit control table; they do not use an ASN. The designated ASN-second-table entry is normally a pseudo ASN-second-table entry (one not in the structure able to be indexed by means of an ASN).

ASN-Second-Table-Entry Instance Number and ASN Reuse

The ASN-and-LX-reuse facility may be installed on the model. If this facility is installed, the ASN-second-table entry contains an ASN-second-table-entry instance number (ASTEIN), and certain new definitions related to the ASTEIN and to a new linkage-second-table-entry sequence number (LSTESN) apply. The definitions related to the ASTEIN are summarized below and also given in the appropriate sections of this publication.

- The facility also includes the following new control bits:
  - The ASN-and-LX-reuse control (R), bit 44 of control register 0
  - The controlled-ASN bit (CA), bit 30 of word 1 of the ASN-second-table entry
  - The reusable-ASN bit (RA), bit 31 of word 1 of the ASN-second-table entry

The three control bits are shown as follows:

Control Register 0

<table>
<thead>
<tr>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
</tr>
</tbody>
</table>

Word 1 of ASN-Second-Table Entry

<table>
<thead>
<tr>
<th>AX</th>
<th>ATL</th>
<th>CA</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

- The facility also includes the primary ASTEIN in bit positions 0-31 of control register 4 and the secondary ASTEIN in bit positions 0-31 of control register 3. The primary ASTEIN is a copy of the ASTEIN in the ASN-second-table entry for the current primary address space specified by the PASN in control register 4, and the secondary ASTEIN is a copy of the ASTEIN in the ASN-second-table entry for the current secondary address space specified by the SASN in control register 3. The complete formats of control registers 3 and 4 are as follows:

Control Register 3

<table>
<thead>
<tr>
<th>SASTEIN</th>
<th>PKM</th>
<th>SASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control Register 4

<table>
<thead>
<tr>
<th>PASTEIN</th>
<th>AX</th>
<th>PASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The following operations are performed if the ASN-and-LX-reuse control, bit 44 of control register 0, is one. When LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, or SET SECONDARY ASN
(1) sets the primary ASN in control register 4 or the secondary ASN in control register 3 equal to a specified ASN and (2) sets the primary address-space-control element in control register 1 or the secondary address-space-control element in control register 7, respectively, with the value of an address-space-control element obtained from an ASN-second-table entry located by means of ASN translation of the specified ASN (or located, by PROGRAM CALL only, by means of an ASTE address), the instruction also sets the primary ASTEIN in control register 4 or secondary ASTEIN in control register 3, respectively, with the value of the ASTEIN in the ASN-second-table entry. However, PROGRAM TRANSFER and SET SECONDARY ASN, which are performing their space-switching operations, recognize a special-operation exception if the reusable-ASN bit, bit 31 of word 1, in the ASN-second-table entry is one.

When the ASN-and-LX-reuse control is one and any of the instructions named above or BRANCH IN SUBSPACE GROUP sets the secondary ASN and secondary address-space-control element equal to the primary ASN (in the space-switching stacking PROGRAM CALL operation, this may be the old or the new primary ASN, as determined by a bit in the entry-table entry used) and primary address-space-control element, respectively, it also sets the secondary ASTEIN in control register 3 equal to the primary ASTEIN in control register 4 (again, in the space-switching stacking PROGRAM CALL case, this may be the old or the new primary ASTEIN). Since this function does not include the accessing of an ASN-second-table entry, it is not affected by a reusable-ASN bit.

PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE operate as described above except (1) independent of the value of the ASN-and-LX-reuse control (they always update the primary ASTEIN and secondary ASTEIN as described above), (2) without recognizing an exception due to the reusable-ASN bit (they ignore the bit), and (3) in their space-switching forms and the problem state at the beginning of the operation, by recognizing a special-operation exception if the controlled-ASN bit, bit 30 of word 1, is one in the ASN-second-table entry located by ASN translation. PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE ignore the controlled-ASN bit if the CPU is in the supervisor state at the beginning of the operation. (PROGRAM TRANSFER WITH INSTANCE may switch the CPU from the supervisor state to the problem state.)

- PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE operate the same as PROGRAM TRANSFER and SET SECONDARY ASN, respectively, except as described in the preceding item and except as follows.

Independent of the ASN-and-LX-reuse control and the reusable-ASN bit, but not of the controlled-ASN bit, each of PROGRAM TRANSFER WITH INSTANCE with space switching and SET SECONDARY ASN WITH INSTANCE with space switching, after it has used the ASN in the R₁ general register to locate an ASN-second-table entry, compares an ASTEIN in bit positions 0-31 of general register R₁ to the ASTEIN in the entry. The two ASTEINs must be equal; otherwise, an ASTE-instance exception is recognized. This comparison is performed in both the problem and the supervisor state.

- The following operations are performed if the ASN-and-LX-reuse control is one. Each of stacking PROGRAM CALL and BRANCH AND STACK places the current primary ASTEIN in bytes 180-183 of the linkage-stack state entry that it forms, and it places the current secondary ASTEIN, bits 0-31 of control register 3, in bytes 176-179 of the state entry.

Note: There is not a test of an ASTEIN related to the ASN in the entry-table entry used by PROGRAM CALL. There may be a test of a linkage-second-table-entry sequence number related to the linkage index used to locate the entry-table entry, as summarized in "ASN-and-LX-Reuse Control (R)" on page 5-32 in the section “PC-Number Translation.”

- The following operations are performed if the ASN-and-LX-reuse control is one. PROGRAM RETURN with space switching, after it has used the PASN in bytes 134 and 135 of the
linkage-stack program-call state entry to locate an ASN-second-table entry (because that PASN is not equal to the current PASN in control register 4), compares the primary ASTEIN in bytes 180-183 of the state entry to the ASTEIN in the ASN-second-table entry. PROGRAM RETURN to current primary or with space switching, if it uses the SASN in bytes 130 and 131 of the state entry to locate an ASN-second-table entry (because that SASN is not equal to the new PASN), compares the secondary ASTEIN in bytes 176-179 of the state entry to the ASTEIN in the ASN-second-table entry. The one or two comparisons of ASTEINs must each give equal results; otherwise, an ASTE-instance exception is recognized. These operations occur independent of the controlled-ASN and reusable-ASN bits in the ASN-second-table entry. PROGRAM RETURN does not compare ASTEINs when it unstacks a branch state entry.

The following format applies, and the operations are performed, if the ASN-and-LX-reuse control is one. The first operand of LOAD ADDRESS SPACE PARAMETERS is two consecutive doublewords having the following format:

```
First Operand of LOAD ADDRESS SPACE PARAMETERS

<table>
<thead>
<tr>
<th>SASTEIN-d</th>
<th>PKM-d</th>
<th>SASN-d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PASTEIN-d</th>
<th>AX-d</th>
<th>PASN-d</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>96</td>
<td>112</td>
</tr>
</tbody>
</table>
```

If PASN translation is performed to locate an ASN-second-table entry, PASTEIN-d is compared to the ASTEIN in the ASN-second-table entry, and they must be equal; otherwise, condition code 1 is set. If SASN translation is performed to locate an ASN-second-table entry, SASTEIN-d is compared to the ASTEIN in the ASN-second-table entry, and they must be equal; otherwise, condition code 2 is set. These operations are performed independent of the controlled-ASN and reusable-ASN bits in the ASN-second-table entries.

Independent of the ASN-and-LX-reuse control, EXTRACT PRIMARY ASN AND INSTANCE places the current primary ASN, bits 48-63 of control register 4, in bit positions 48-63 of general register R; places the current primary ASTEIN, bits 0-31 of control register 4, in bit positions 0-31 of the general register; and places zeros in bit positions 32-47 of the general register. EXTRACT SECONDARY ASN AND INSTANCE operates the same except that it obtains the secondary ASN and secondary ASTEIN in control register 3 for placement in the general register.

Independent of the ASN-and-LX-reuse control, a new code, code 5, of the EXTRACT STACKED STATE instruction is valid. Code 5 causes the saved secondary ASTEIN, bytes 176-179 of the state entry, to be placed in bit positions 0-31 of the R general register and the saved primary ASTEIN, bytes 180-183 of the state entry, to be placed in bit positions 0-31 of general register R + 1. Bits 32-63 of the general registers remain unchanged.

These uses of the ASTEIN allow, in all cases except one, an ASN associated with a particular ASTEIN to be made unusable when the ASN and the ASN-second-table entry it designates are reasigned to specify a conceptually different address space. The ASN-and-ASTEIN combination can be made unusable by (1) setting to one the reusable-ASN bit in the ASN-second-table entry, which prevents the use of PROGRAM TRANSFER and SET SECONDARY ASN, and (2) changing the ASTEIN in the ASN-second-table entry, which prevents the use of LOAD ADDRESS SPACE PARAMETERS, PROGRAM TRANSFER WITH INSTANCE, PROGRAM RETURN, and SET SECONDARY ASN WITH INSTANCE. The uncovered case is that of an ASN (and the corresponding ASN-second-table-entry address) in an entry-table entry, and this case should be handled by means of deletion, by the control program, of the entry-table entry. However, if the control program then reforms the entry-table entry (forms an entry-table entry located by means of the same PC number, used by PROGRAM CALL, as was the deleted entry-table entry), a PC number that was used to link to the conceptually deleted address space will then be usable to link to a different (different ASN) or conceptually different (same ASN but different contents) address space, and this error should be avoided by using the linkage-second-table-entry sequence number as described in Chapter 3. Storage 3-21
Programming Notes:

1. The reusable-ASN bit of the ASN-and-LX-reuse facility provides reliability, availability, and serviceability. Most importantly, it provides availability since it allows ASNs to be reused. However, it does not provide system integrity since the ASTEIN in the general register used by PROGRAM TRANSFER WITH INSTANCE or SET SECONDARY ASN WITH INSTANCE is provided by the program. (The authorization index used by those instructions normally provides system integrity but may fail to do so if an ASN authorized by an unchanged authorization index is reused.) The controlled-ASN bit provides system integrity since, if the CPU is in the problem state at the beginning of the operation, PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE are unable to proceed successfully after accessing an ASN-second-table entry in which the controlled-ASN bit is one. When the calling program and called program are both executed in the problem state, a program in an address space for which the controlled-ASN bit is one should perform linkages to programs in other address spaces only by means of stacking PROGRAM CALL. If that is done, the return by means of PROGRAM RETURN will always be successful, or will fail appropriately when the ASN of the first address space has been changed, because of the comparison to the saved primary ASTEIN in the linkage-stack state entry.

2. A program given control by a basic PROGRAM CALL operation can use EXTRACT SECONDARY ASN AND INSTANCE to obtain the ASTEIN to be used by PROGRAM TRANSFER WITH INSTANCE to return to the calling program or by SET SECONDARY ASN WITH INSTANCE to restore its secondary address space after a change of that space. This EXTRACT SECONDARY ASN AND INSTANCE instruction should be executed while the original secondary space remains continuously the secondary space; otherwise, depending on actions by the control program, EXTRACT SECONDARY ASN AND INSTANCE may return an ASTEIN that allows return to or use of a conceptually incorrect secondary space for which the ASTEIN has been changed.

3. A summary of the functions related to ASN reuse is given in Figure 3-6 on page 3-23.

4. There are certain programming errors, or situations that are not necessarily errors, that will not be detected. These cases are described in the definitions of the named instructions and are as follows:
   - In LOAD ADDRESS SPACE PARAMETERS:
     - SASN translation is performed only when, but not necessarily when, SASN-d is not equal to PASN-d. When SASN-d is equal to PASN-d, SASCE-new and SASTEIN-new are set equal to PASCE-new and PASTEIN-new, respectively. In this case, there is not a test of whether SASTEIN-d is equal to PASTEIN-d; SASTEIN-d is ignored.
     - When SASN-d is not equal to PASN-d and is equal to SASN-old, bit 61 (force ASN translation) is zero, and bit 63 (skip SASN authorization) is one, SASN translation is not performed, and SASCE-old and SASTEIN-old become SASCE-new and SASTEIN-new, respectively. In this case, there is not a test of whether SASTEIN-d is equal to SASTEIN-old; SASTEIN-d is ignored.
   - In PROGRAM RETURN, if the new SASN is equal to the new PASN, the SASCE in control register 7 is set equal to the new PASCE in control register 1. The SASTEIN, PKM, and SASN in control register 3 remain as restored from the state entry. In this case, there is not a test of whether the new SASTEIN is equal to the new PASTEIN. (There has already been a test of whether the PASTEIN saved in the entry, which becomes the new PASTEIN, equals the ASTEIN in the new PASTE.)
   - In the PROGRAM TRANSFER WITH INSTANCE to-current-primary operation, there is not a test of whether the current PASTEIN equals the ASTEIN specified in
bit positions 0-31 of general register $R_i$; the ASTEIN is ignored.

- In the SET SECONDARY ASN WITH INSTANCE to-current-primary operation, there is not a test of whether the current PASTEIN (to which the SASTEIN is set equal) equals the ASTEIN specified in bit positions 0-31 of general register $R_i$; the ASTEIN is ignored.

<table>
<thead>
<tr>
<th>Function</th>
<th>Required Bit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LASP, PC, and PR update PASTEIN or SASTEIN in a control register after accessing an ASN-second-table entry</td>
<td>1 - -</td>
</tr>
<tr>
<td>PT and SSAR update PASTEIN or SASTEIN in a control register after accessing an ASN-second-table entry</td>
<td>1 - 0(^1)</td>
</tr>
<tr>
<td>BSG, LASP, PC, PR, PT, and SSAR set SASTEIN equal to PASTEIN</td>
<td>1 - -</td>
</tr>
<tr>
<td>PTI and SSAIR update PASTEIN or SASTEIN in a control register after accessing an ASN-second-table entry</td>
<td>- 0(^2) -</td>
</tr>
<tr>
<td>PTI and SSAIR set SASTEIN equal to PASTEIN</td>
<td>- - -</td>
</tr>
<tr>
<td>PTI and SSAIR, after accessing an ASN-second-table entry, compare PASTEIN or SASTEIN in the state entry to ASTEIN in the entry(^3)</td>
<td>- 0(^2) -</td>
</tr>
<tr>
<td>Stacking PC and BAKR copy PASTEIN and SASTEIN from control registers to state entry</td>
<td>1 - -</td>
</tr>
<tr>
<td>PR, after accessing an ASN-second-table entry, compares PASTEIN or SASTEIN in the state entry to ASTEIN in the ASN-second-table entry(^3)</td>
<td>1 - -</td>
</tr>
<tr>
<td>LASP, after accessing an ASN-second-table entry, compares PASTEIN-d or SASTEIN-d to ASTEIN in the entry(^4)</td>
<td>1 - -</td>
</tr>
<tr>
<td>EPAIR copies PASTEIN and PASN from control register 4 to general register $R_i$</td>
<td>- - -</td>
</tr>
<tr>
<td>ESAIR copies SASTEIN and SASN from control register 3 to general register $R_i$</td>
<td>- - -</td>
</tr>
</tbody>
</table>

Figure 3-6 (Part 1 of 2). Summary of Functions Related to ASN Reuse
### Function Summary of Functions Related to ASN Reuse

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESTA code 5 copies PASTEIN and SASTEIN from state entry to general registers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Required Bit Value</th>
<th>R</th>
<th>CA</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### Explanation:
- Bit is ignored or not applicable to the operation.

1. A special-operation exception is recognized if the bit is one.

2. A special-operation exception is recognized if the bit is one and the CPU is in the problem state at the beginning of the operation.

3. An ASTE-instance exception is recognized if the ASTEINs are not equal.

4. Condition code 1 (if PASTEIN comparison) or 2 (if SASTEIN comparison) is set if the ASTEIN is not equal.

- CA: Controlled-ASN bit, bit 30 of word 1 of ASN-second-table entry.
- R: ASN-and-LX-reuse control, bit 44 of control register 0.
- RA: Reusable-ASN bit, bit 31 of word 1 of ASN-second-table entry.

---

### ASN Translation

ASN translation is the process of translating a 16-bit ASN to locate the ASN-second-table entry designated by the ASN. ASN translation is performed as part of PROGRAM TRANSFER with space switching (PT-ss) PROGRAM TRANSFER WITH INSTANCE with space switching (PTI-ss), SET SECONDARY ASN with space switching (SSAR-ss), and SET SECONDARY ASN WITH INSTANCE with space switching (SSAIR-ss), and it may be performed as part of LOAD ADDRESS SPACE PARAMETERS. For PT-ss and PTI-ss the ASN which is translated replaces the primary ASN in control register 4. For SSAR-ss and SSAIR-ss, the ASN which is translated replaces the secondary ASN in control register 3. These two translation processes are called primary ASN translation and secondary ASN translation, respectively, and both can occur for LOAD ADDRESS SPACE PARAMETERS. The ASN-translation process is the same for both primary and secondary ASN translation; only the uses of the results of the process are different.

ASN translation may also be performed as part of PROGRAM RETURN. Primary ASN translation is performed as part of PROGRAM RETURN with space switching (PR-ss). Secondary ASN translation is performed if the secondary ASN restored by PROGRAM RETURN (PR-ss or PROGRAM RETURN to current primary) does not equal the primary ASN restored by PROGRAM RETURN.

PROGRAM CALL with space switching (PC-ss) performs the equivalent of primary ASN translation by obtaining a primary ASN and the address of the corresponding ASN-second-table entry from an entry-table entry.

The ASN-translation process uses two tables, the ASN first table and the ASN second table. They are used to locate the ASN-second-table entry and a third table, the authority table, which is used when ASN authorization is performed.

For the purposes of this translation, the 16-bit ASN is considered to consist of two parts: the ASN-first-table index (AFX) is the leftmost 10 bits of the ASN, and the ASN-second-table index (ASX) is the six rightmost bits. The ASN has the following format:

```plaintext
ASN

<table>
<thead>
<tr>
<th>AFX</th>
<th>ASX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
```

The AFX is used to select an entry from the ASN first table. The origin of the ASN first table is designated by the ASN-first-table origin in control register 14. The ASN-first-table entry contains the origin of the ASN second table. The ASX is used to select an entry from the ASN second table.

As a result of primary ASN translation and during the operation of PROGRAM CALL with space switching, the address of the located ASN-second-table entry (ASTE) is placed in
control register 5 as the new primary-ASTE origin (PASTEO).

ASN-Translation Controls

ASN translation is controlled by the ASN-translation-control bit and the ASN-first-table origin, both of which reside in control register 14.

Control Register 14

<table>
<thead>
<tr>
<th>T</th>
<th>AFTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>63</td>
</tr>
</tbody>
</table>

ASN-Translation Control (T): Bit 44 of control register 14 is the ASN-translation-control bit. This bit provides a mechanism whereby the control program can indicate whether ASN translation can occur while a particular program is being executed, and also whether the execution of PROGRAM CALL with space switching is allowed. Bit 44 must be one to allow completion of these instructions:

- LOAD ADDRESS SPACE PARAMETERS
- PROGRAM CALL with space switching
- PROGRAM RETURN with space switching or when the restored SASN does not equal the restored PASN
- PROGRAM TRANSFER with space switching
- PROGRAM TRANSFER WITH INSTANCE with space switching
- SET SECONDARY ASN
- SET SECONDARY ASN WITH INSTANCE

Otherwise, a special-operation exception is recognized. The ASN-translation-control bit is examined in both the problem and the supervisor states.

ASN-First-Table Origin (AFTO): Bits 45-63 of control register 14, with 12 zeros appended on the right, form a 31-bit real address that designates the beginning of the ASN first table.

ASN-Translation Tables

The ASN-translation process consists in a two-level lookup using two tables: an ASN first table and an ASN second table. These tables reside in real storage.

ASN-First-Table Entries

An entry in the ASN first table has the following format:

- | ASTO |
  - | 26 31 |

The fields in the entry are allocated as follows:

- **AFX-Invalid Bit (I):** Bit 0 controls whether the ASN second table associated with the ASN-first-table entry is available. When bit 0 is zero, ASN translation proceeds by using the designated ASN second table. When the bit is one, the ASN translation cannot continue.

- **ASN-Second-Table Origin (ASTO):** Bits 1-25, with six zeros appended on the right, are used to form a 31-bit real address that designates the beginning of the ASN second table.

ASN-Second-Table Entries

The ASN-second-table entry has a length of 64 bytes, with only the first 48 bytes currently in use. Bytes 0-47 of the entry have the following format:

- | ATO | B |
  - | 30 31 |
- | AX | ATL | C | R |
  - | 60 63 |
- | RTO (RSTKO) Part 1— |
  - RTO, STO, or RSTKO 64 95 |
- | RTO/STO (Cont.) | GPSX | R | DT | TL |
  - | R=0 96 115 118 122 124 127 |
- | RSTKO (Cont.) | GPSX | R |
  - | R=1 96 115 118 122 127 |
The fields in bytes 0-47 of the ASN-second-table entry are allocated as follows. Only the fields that are used in or as a result of ASN translation or PROGRAM CALL with space switching are described in detail.

**ASX-Invalid Bit (I):** Bit 0 controls whether the address space associated with the ASN-second-table entry is available. When bit 0 is zero, ASN translation proceeds. When the bit is one, the ASN translation cannot continue.

**Authority-Table Origin (ATO):** Bits 1-29, with two zeros appended on the right, are used to form a 31-bit real address that designates the beginning of the authority table.

**Base-Space Bit (B):** Bit 31 specifies, when one, that the address space associated with the ASN-second-table entry is the base space of a subspace group. Bit 31 is further described in [Subspace-Group ASN-Second-Table Entries](page 5-66) on page 5-66.

**Authorization Index (AX):** Bits 32-47 are used in ASN authorization as an index to locate the authority bits in the authority table. The AX field is used as a result of primary ASN translation by PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, and, possibly, LOAD ADDRESS SPACE PARAMETERS. It is also used by PROGRAM CALL with space switching. The AX field is ignored after secondary ASN translation.

**Authority-Table Length (ATL):** Bits 48-59 specify the length of the authority table in units of four bytes, thus making the authority table variable in multiples of 16 entries. The length of the authority table, in units of four bytes, is one more than the ATL value. The contents of the ATL field are used to establish whether the entry designated by a particular AX falls within the authority table.

**Controlled-ASN Bit (CA):** PROGRAM TRANSFER WITH INSTANCE with space switching and SET SECONDARY ASN WITH INSTANCE with space switching recognize a special-operation exception if bit 62 is one and the CPU is in the problem state at the beginning of the operation. Bit 62 is ignored in the supervisor state.

**Reusable-ASN Bit (RA):** If the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, PROGRAM TRANSFER with space switching and SET SECONDARY ASN with space switching recognize a
special-operation exception if bit 63 is one in the problem or the supervisor state.

**Address-Space-Control Element (ASCE):** Bits 64-127 are an eight-byte address-space-control element (ASCE) that may be a region-table designation (RTD), a segment-table designation (STD), or a real-space designation (RSD). (The term “region-table designation” is used to mean a region-first-table designation, region-second-table designation, or region-third-table designation.) The ASCE field is used as a result of ASN translation or in PROGRAM CALL with space switching to replace the primary ASCE (PASCE) or the secondary ASCE (SASCE). For PROGRAM CALL with space switching, the ASCE field replaces the PASCE, bits 0-63 of control register 1. For SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE, the ASCE field replaces the SASCE, bits 0-63 of control register 7. Each of these actions may occur independently for LOAD ADDRESS SPACE PARAMETERS. For PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, the ASCE field replaces both the PASCE and the SASCE. For PROGRAM RETURN, as a result of primary ASN translation, the ASCE field replaces the PASCE, and, as a result of secondary ASN translation, the ASCE field replaces the SASCE. The contents of the entire ASCE field are placed in the appropriate control registers without being inspected for validity.

The subspace-group-control bit (G), bit 118 of the ASCE field, indicates, when one, that the ASCE specifies an address space that is the base space or a subspace of a subspace group. The bit is further described in "Subspace-Group ASN-Second-Table Entries" on page 5-66.

Bit 121 (X) of the ASCE field is the space-switch-event-control bit. When, in the space-switching operations of PROGRAM CALL, PROGRAM RETURN, and PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, this bit is one in control register 1 either before or after the execution of the instruction, a program interruption for a space-switch event occurs after the execution of the instruction is completed. A space-switch-event program interruption also occurs after the completion of a SET ADDRESS SPACE CONTROL, SET ADDRESS SPACE CONTROL FAST, or RESUME PROGRAM instruction that changes the translation mode either to or from the home-space mode when this bit is one in either control register 1 or control register 13. When, in LOAD ADDRESS SPACE PARAMETERS, this fact is indicated by the condition code.

The real-space-control bit (R), bit 122 of the ASCE field, indicates, when zero, that the ASCE is a region-table or segment-table designation or, when one, that the ASCE is a real-space designation.

When bit 122 is zero, the designation-type-control bits (DT), bits 124 and 125 of the ASCE field, indicate the designation type of the ASCE. A value 11, 10, 01, or 00 binary of bits 124 and 125 indicates a region-first-table designation, region-second-table designation, region-third-table designation, or segment-table designation, respectively.

The other fields in the ASCE (RTO, STO, P, S, TL, and RSTKO) are described in "Control Register 1" on page 3-36.

**Access-List Designation (ALD):** The access-list-designation (ALD) field is described in "ASN-Second-Table Entries" on page 5-55.

**ASN-Second-Table-Entry Sequence Number (ASTESEN):** The ASTE-sequence-number (ASTESEN) field is described in "ASN-Second-Table Entries" on page 5-55.

**Linkage-Table Designation (LTD) or Linkage-First-Table Designation (LFTD):** The linkage-table-designation (LTD) or linkage-first-table designation (LFTD) field in the ASN-second-table entry is described in "PC-Number Translation Control" on page 5-32.

**ASN-Second-Table-Entry Instance Number (ASTEIN):** When the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control in control register 0, bits 352-383 are compared to an ASTEIN specified along with an ASN for use by PROGRAM RETURN or LOAD ADDRESS SPACE PARAMETERS. The ASTEINs must be equal; otherwise, an ASTE-instance exception is recognized by PROGRAM RETURN or condition code 1 or 2 is set by LOAD ADDRESS SPACE PARAMETERS. This comparison and the exception result also occur in the operations of PROGRAM TRANSFER.
With Instance and Set Secondary ASN
With Instance except independent of the
ASN-and-LX-reuse control.

Bits 224-319 in the ASN-second-table entry are
available for use by programming.

Programming Note: All unused fields in the
ASN-second-table entry, including the unused
fields in bytes 0-31 and all of bytes 40-43 and
48-63 should be set to zeros. These fields are
reserved for future extensions, and programs
which place nonzero values in these fields may
not operate compatibly on future machines.

ASN-Translation Process

This section describes the ASN-translation
process as it is performed during the execution of
the space-switching forms of PROGRAM
RETURN, PROGRAM TRANSFER, PROGRAM
TRANSFER WITH INSTANCE, SET SECONDARY ASN,
and SET SECONDARY ASN
WITH INSTANCE, and also in PROGRAM
RETURN when the restored secondary ASN does
not equal the restored primary ASN. ASN trans-
lation for LOAD ADDRESS SPACE PARAME-
TERS is the same except that AFX-translation and
ASX-translation exceptions do not occur; such
conditions are instead indicated by the condition
code. Translation of an ASN is performed by
means of two tables, an ASN first table and an
ASN second table, both of which reside in main
storage.

The ASN first index is used to select an entry from
the ASN first table. This entry designates the
ASN second table to be used.

The ASN second index is used to select an entry
from the ASN second table.

If the I bit is one in either the ASN-first-table entry
or the ASN-second-table entry, the entry is invalid,
and the ASN-translation process cannot be com-
pleted. An AFX-translation exception or
ASX-translation exception is recognized.

Whenever access to main storage is made during
the ASN-translation process for the purpose of
fetching an entry from an ASN first table or ASN
second table, key-controlled protection does not
apply.

The ASN-translation process is shown in
Figure 3-7 on page 3-29.
Figure 3-7. ASN Translation

**ASN-First-Table Lookup**

The AFX portion of the ASN, in conjunction with the ASN-first-table origin, is used to select an entry from the ASN first table.

The 31-bit real address of the ASN-first-table entry is obtained by appending 12 zeros on the right to the AFT origin contained in bit positions 45-63 of control register 14 and adding the AFX portion with two rightmost and 19 leftmost zeros appended. This addition cannot cause a carry into bit position 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All four bytes of the ASN-first-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the ASN-first-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the four-byte AFT entry specifies whether the corresponding AST is available. If this bit is one, an AFX-translation exception is recognized. The entry fetched from the AFT is used to access the AST.

**ASN-Second-Table Lookup**

The ASX portion of the ASN, in conjunction with the ASN-second-table origin contained in the ASN-first-table entry, is used to select an entry from the ASN second table.

The 31-bit real address of the ASN-second-table entry is obtained by appending six zeros on the right to bits 1-25 of the ASN-first-table entry and adding the ASX with six rightmost and 19 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{31}$ - 1 to zero. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

The fetch of the 64 bytes of the ASN-second-table entry appears to be word concurrent as observed...
by other CPUs, with the leftmost word fetched first. The order in which the remaining 15 words are fetched is unpredictable. The fetch access is not subject to protection. When the storage address which is generated for fetching the ASN-second-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the ASN-second-table entry specifies whether the address space is accessible. If this bit is one, an ASX-translation exception is recognized.

Recognition of Exceptions during ASN Translation

The exceptions which can be encountered during the ASN-translation process are collectively referred to as ASN-translation exceptions. A list of these exceptions and their priorities is given in Chapter 6, “Interruptions.”

ASN Authorization

ASN authorization is the process of testing whether the program associated with the current authorization index is permitted to establish a particular address space. The ASN authorization is performed as part of PROGRAM TRANSFER with space switching (PT-ss) and SET SECONDARY ASN with space switching (SSAR-ss) and may be performed as part of LOAD ADDRESS SPACE PARAMETERS. ASN authorization is performed after the ASN-translation process for these instructions.

ASN authorization is also performed as part of PROGRAM RETURN when the restored secondary ASN does not equal the restored primary ASN. ASN authorization of the restored secondary ASN is performed after ASN translation of the restored secondary ASN.

When performed as part of PT-ss, the ASN authorization tests whether the ASN can be established as the primary ASN and is called primary-ASN authorization. When performed as part of LOAD ADDRESS SPACE PARAMETERS, PROGRAM RETURN, or SSAR-ss, the ASN authorization tests whether the ASN can be established as the secondary ASN and is called secondary-ASN authorization.

The ASN authorization is performed by means of an authority table in real storage which is designated by the authority-table-origin and authority-table-length fields in the ASN-second-table entry.

ASN-Authorization Controls

ASN authorization uses the authority-table origin and the authority-table length from the ASN-second-table entry, together with an authorization index.

Control Register 4

For PT-ss and SSAR-ss, the current contents of control register 4 include the authorization index. For LOAD ADDRESS SPACE PARAMETERS and PROGRAM RETURN, the value which will become the new contents of control register 4 is used. The register has the following format:

<table>
<thead>
<tr>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

Authorization Index (AX): Bits 32-47 of control register 4 are used as an index to locate the authority bits in the authority table.

ASN-Second-Table Entry

The ASN-second-table entry which is fetched as part of the ASN translation process contains information which is used to designate the authority table. An entry in the ASN second table has the following format:

<table>
<thead>
<tr>
<th>ATO</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

Authority-Table Origin (ATO): Bits 1-29, with two zeros appended on the right, are used to form a 31-bit real address that designates the beginning of the authority table.

Authority-Table Length (ATL): Bits 48-59 specify the length of the authority table in units of four bytes, thus making the authority table variable.
in multiples of 16 entries. The length of the authority table, in units of four bytes, is equal to one more than the ATL value. The contents of the length field are used to establish whether the entry designated by the authorization index falls within the authority table.

**Authority-Table Entries**

The authority table consists of entries of two bits each; accordingly, each byte of the authority table contains four entries in the following format:

```
| PS | PS | PS | PS |
```

0 7

The fields are allocated as follows:

**Primary Authority (P):** The left bit of an authority-table entry controls whether the program with the authorization index corresponding to the entry is permitted to establish the address space as a primary address space. If the P bit is one, the establishment is permitted. If the P bit is zero, the establishment is not permitted.

**Secondary Authority (S):** The right bit of an authority-table entry controls whether the program with the corresponding authorization index is permitted to establish the address space as a secondary address space. If the S bit is one, the establishment is permitted. If the S bit is zero, the establishment is not permitted.

The authority table is also used in the extended-authorization process, as part of access-register translation. Extended authorization is described in "Authorizing the Use of the Access-List Entry on page 5-61."

**ASN-Authorization Process**

This section describes the ASN-authorization process as it is performed during the execution of PROGRAM TRANSFER with space switching and SET SECONDARY ASN with space switching. For these two instructions, the ASN-authorization process is performed by using the authorization index currently in control register 4. Secondary authorization for PROGRAM RETURN, when the restored secondary ASN does not equal the restored primary ASN, and for LOAD ADDRESS SPACE PARAMETERS is the same, except that the value which will become the new contents of control register 4 is used for the authorization index. Also, for LOAD ADDRESS SPACE PARAMETERS, a secondary-authority exception does not occur. Instead, such a condition is indicated by the condition code.

The ASN-authorization process is performed by using the authorization index, in conjunction with the authority-table origin and length from the AST entry, to select an authority-table entry. The entry is fetched, and either the primary- or the secondary-authority bit is examined, depending on whether the primary- or secondary-ASN-authorization process is being performed. The ASN-authorization process is shown in Figure 3-8 on page 3-32.
Authority-Table Lookup
The authorization index, in conjunction with the authority-table origin contained in the ASN-second-table entry, is used to select an entry from the authority table.

The authorization index is contained in bit positions 32-47 of control register 4.

Bit positions 1-29 of the AST entry contain the leftmost 29 bits of the 31-bit real address of the authority table (ATO), and bit positions 48-59 contain the length of the authority table (ATL).

The 31-bit real address of a byte in the authority table is obtained by appending two zeros on the right to the authority-table origin and adding the 14 leftmost bits of the authorization index with 17 zeros appended on the left. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{31} - 1$ to zero. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the authority-table-entry-lookup process, bits 0-11 of the authorization index are compared against the authority-table length. If the compared portion is greater than the authority-table length, a primary-authority exception or secondary-authority exception is recognized for PT-ss or SSAR-ss, respectively. For LOAD ADDRESS SPACE PARAMETERS, when the authority-table length is exceeded, condition code 2 is set.

R: Address is real
*: Last 48 bytes of ASTE are not shown

Figure 3-8. ASN Authorization
The fetch access to the byte in the authority table is not subject to protection. When the storage address which is generated for fetching the byte designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

The byte contains four authority-table entries of two bits each. The rightmost two bits of the authorization index, bits 46 and 47 of control register 4, are used to select one of the four entries. The left or right bit of the entry is then tested, depending on whether the authorization test is for a primary ASN or a secondary ASN. The following table shows the bit which is selected from the byte as a function of bits 46 and 47 of the authorization index and the instruction PT-ss, SSAR-ss, PROGRAM RETURN, or LOAD ADDRESS SPACE PARAMETERS.

<table>
<thead>
<tr>
<th>Authorization-Index Bits</th>
<th>Bit Selected from Authority-Table Byte for Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>46 47</td>
<td>P Bit (PT-ss)</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>2</td>
</tr>
<tr>
<td>1 0</td>
<td>4</td>
</tr>
<tr>
<td>1 1</td>
<td>6</td>
</tr>
</tbody>
</table>

If the selected bit is one, the ASN is authorized, and the appropriate fields in the AST entry are loaded into the appropriate control registers. If the selected bit is zero, the ASN is not authorized, and a primary-authority exception is recognized for PT-ss or a secondary-authority exception is recognized for SSAR-ss or PROGRAM RETURN. For LOAD ADDRESS SPACE PARAMETERS, when the ASN is not authorized, condition code 2 is set.

**Recognition of Exceptions during ASN Authorization**

The exceptions which can be encountered during the primary- and secondary-ASN-authorization processes and their priorities are described in the definitions of the instructions in which ASN authorization is performed.

**Program Note**: The primary- and secondary-authority exceptions cause nullification in order to permit dynamic modification of the authority table. Thus, when an address space is created or “swapped in,” the authority table can first be set to all zeros and the appropriate authority bits set to one only when required.

**Dynamic Address Translation**

Dynamic address translation (DAT) provides the ability to interrupt the execution of a program at an arbitrary moment, record it and its data in auxiliary storage, such as a direct-access storage device, and at a later time return the program and the data to different main-storage locations for resumption of execution. The transfer of the program and its data between main and auxiliary storage may be performed piecemeal, and the return of the information to main storage may take place in response to an attempt by the CPU to access it at the time it is needed for execution. These functions may be performed without change or inspection of the program and its data, do not require any explicit programming convention for the relocated program, and do not disturb the execution of the program except for the time delay involved.

With appropriate support by an operating system, the dynamic-address-translation facility may be used to provide to a user a system wherein storage appears to be larger than the main storage which is available in the configuration. This apparent main storage is referred to as virtual storage, and the addresses used to designate locations in the virtual storage are referred to as virtual addresses. The virtual storage of a user may far exceed the size of the main storage which is available in the configuration and normally is maintained in auxiliary storage. The virtual storage is considered to be composed of blocks of addresses, called pages. Only the most recently referred-to pages of the virtual storage are assigned to occupy blocks of physical main storage. As the user refers to pages of virtual storage that do not appear in main storage, they are brought in to replace pages in main storage that are less likely to be needed. The swapping of pages of storage may be performed by the operating system without the user's knowledge.

The sequence of virtual addresses associated with a virtual storage is called an address space. With appropriate support by an operating system, the
dynamic-address-translation facility may be used to provide a number of address spaces. These address spaces may be used to provide degrees of isolation between users. Such support can consist of a completely different address space for each user, thus providing complete isolation, or a shared area may be provided by mapping a portion of each address space to a single common storage area. Also, instructions are provided which permit a semiprivileged program to access more than one such address space. Dynamic address translation provides for the translation of virtual addresses from multiple different address spaces without requiring that the translation parameters in the control registers be changed. These address spaces are called the primary address space, secondary address space, and AR-specified address spaces. A privileged program can also cause the home address space to be accessed.

In the process of replacing blocks of main storage by new information from an external medium, it must be determined which block to replace and whether the block being replaced should be recorded and preserved in auxiliary storage. To aid in this decision process, a reference bit and a change bit are associated with the storage key. Dynamic address translation may be specified for instruction and data addresses generated by the CPU but is not available for the addressing of data and of CCWs and IDAWs in I/O operations. The CCW-indirect-data-addressing facility is provided to aid I/O operations in a virtual-storage environment.

Address computation can be carried out in the 24-bit, 31-bit, or 64-bit addressing mode. When address computation is performed in the 24-bit or 31-bit addressing mode, 40 or 33 zeros, respectively, are appended on the left to form a 64-bit address. Therefore, the resultant logical address is always 64 bits in length. The real address that is formed by dynamic address translation, and the absolute address that is then formed by prefixing, are always 64 bits in length.

Dynamic address translation is the process of translating a virtual address during a storage reference into the corresponding real address. The virtual address may be a primary virtual address, secondary virtual address, AR-specified virtual address, or home virtual address. These addresses are translated by means of the primary, the secondary, an AR-specified, or the home address-space-control element, respectively. After selection of the appropriate address-space-control element, the translation process is the same for all of the four types of virtual address. An address-space-control element may be a segment-table designation specifying a 2G-byte address space, a region-table designation specifying a 4T-byte, 8P-byte, or 16E-byte space, or a real-space designation specifying a 16E-byte space. (The letters K, M, G, T, P, and E represent kilo, 210, mega, 230, giga, 240, tera, 250, peta, 260, and exa, 260, respectively.) A segment-table designation or region-table designation causes translation to be performed by means of tables established by the operating system in real or absolute storage. A real-space designation causes the virtual address simply to be treated as a real address, without the use of tables in storage.

In the process of translation when using a segment-table designation or a region-table designation, three types of units of information are recognized — regions, segments, and pages. A region is a block of sequential virtual addresses spanning 2G bytes and beginning at a 2G-byte boundary. A segment is a block of sequential virtual addresses spanning 1M bytes and beginning at a 1M-byte boundary. A page is a block of sequential virtual addresses spanning 4K bytes and beginning at a 4K-byte boundary.

The virtual address, accordingly, is divided into four principal fields. Bits 0-32 are called the region index (RX), bits 33-43 are called the segment index (SX), bits 44-51 are called the page index (PX), and bits 52-63 are called the byte index (BX). The virtual address has the following format:

<table>
<thead>
<tr>
<th>RX</th>
<th>SX</th>
<th>PX</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>33</td>
<td>44</td>
<td>52</td>
</tr>
</tbody>
</table>

As determined by its address-space-control element, a virtual address space may be a 2G-byte space consisting of one region, or it may be up to a 16E-byte space consisting of up to 8G regions. The RX part of a virtual address applying to a 2G-byte address space must be all zeros; otherwise, an exception is recognized.

The RX part of a virtual address is itself divided into three fields. Bits 0-10 are called the region
first index (RFX), bits 11-21 are called the region second index (RSX), and bits 22-32 are called the region third index (RTX). Bits 0-32 of the virtual address have the following format:

<table>
<thead>
<tr>
<th>RFX</th>
<th>RSX</th>
<th>RTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>22</td>
</tr>
</tbody>
</table>

A virtual address in which the RTX is the leftmost significant part (a 42-bit address) is capable of addressing 4T bytes (2K regions), one in which the RSX is the leftmost significant part (a 53-bit address) is capable of addressing 8P bytes (4M regions), and one in which the RFX is the leftmost significant part (a 64-bit address) is capable of addressing 16E bytes (8G regions).

A virtual address in which the RX is always zero can be translated into real addresses by means of two translation tables: a segment table and a page table. If the RX may be nonzero, from one to three additional translation tables are required, as follows. If the RFX may be nonzero, a region first table, region second table, and region third table are required. If the RFX is always zero but the RSX may be nonzero, a region second table and region third table are required. If the RFX and RSX are always zero but the RTX may be nonzero, a region third table is required. An exception is recognized if the address-space-control element for an address space does not designate the highest level of table (beginning with the region first table and continuing downward to the segment table) needed to translate a reference to the address space.

A region first table, region second table, or region third table is sometimes referred to simply as a region table. Similarly, a region-first-table designation, region-second-table designation, or region-third-table designation is sometimes referred to as a region-table designation.

The region, segment, and page tables reflect the current assignment of real storage. The assignment of real storage occurs in units of pages, the real locations being assigned contiguously within a page. The pages need not be adjacent in real storage even though assigned to a set of sequential virtual addresses.

To improve performance, translation normally is performed by means of table copies maintained in a special buffer called the translation-lookaside buffer (TLB). The TLB may also contain entries that provide the virtual-equals-real translation specified by a real-space designation.

**Translation Control**

Address translation is controlled by three bits in the PSW and by a set of bits referred to as the translation parameters. The translation parameters are in control registers 0, 1, 7, and 13. Additional controls are located in the translation tables.

Additional controls are provided as described in Chapter 5, “Program Execution.” These controls determine whether the contents of each access register can be used to obtain an address-space-control element for use by DAT.

**Translation Modes**

The three bits in the PSW that control dynamic address translation are bit 5, the DAT-mode bit, and bits 16 and 17, the address-space-control bits. When the DAT-mode bit is zero, then DAT is off, and the CPU is in the real mode. When the DAT-mode bit is one, then DAT is on, and the CPU is in the translation mode designated by the address-space-control bits: 00 designates the primary-space mode, 01 designates the access-register mode, 10 designates the secondary-space mode, and 11 designates the home-space mode. The various modes are shown in Figure 3-9, along with the handling of addresses in each mode.

<table>
<thead>
<tr>
<th>PSW Bit</th>
<th>Handling of Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instruction Addresses</td>
</tr>
<tr>
<td>0 0 0 Off</td>
<td>Real</td>
</tr>
<tr>
<td>0 0 1 Off</td>
<td>Real</td>
</tr>
<tr>
<td>0 1 0 Off</td>
<td>Real</td>
</tr>
<tr>
<td>0 1 1 Off</td>
<td>Primary</td>
</tr>
<tr>
<td>1 0 0 On</td>
<td>Primary</td>
</tr>
<tr>
<td>1 0 1 On</td>
<td>Access-reg</td>
</tr>
<tr>
<td>1 1 0 On</td>
<td>Secondary</td>
</tr>
<tr>
<td>1 1 1 On</td>
<td>Home</td>
</tr>
</tbody>
</table>

**Figure 3-9. Translation Modes**
Control Register 0
One bit is provided in control register 0 for use in controlling dynamic address translation. The bit is assigned as follows:

```
  S
  S
```

37

*Secondary-Space Control (SS):* Bit 37 of control register 0 is the secondary-space-control bit. When this bit is zero and execution of MOVE TO PRIMARY, MOVE TO SECONDARY, or SET ADDRESS SPACE CONTROL is attempted, a special-operation exception is recognized. When this bit is one, it indicates that the region table or segment table designated by the secondary address-space-control element is attached when the CPU is in the primary-space mode.

Control Register 1
Control register 1 contains the primary address-space-control element (PASCE). The register has one of the following two formats, depending on the real-space-control bit (R) in the register:

**Primary Region-Table or Segment-Table Designation (R=0)**

```
Primary Region-Table or Segment-Table Origin
0 52 54 58 60 63
```

**Primary Real-Space Designation (R=1)**

```
Primary Real-Space Token Origin
0 52 54 58 63
```

The fields in the primary address-space-control element are allocated as follows:

**Primary Region-Table or Segment-Table Origin:** Bits 0-51 of the primary region-table or segment-table designation in control register 1, with 12 zeros appended on the right, form a 64-bit address that designates the beginning of the primary region table or segment table. It is unpredictable whether the address is real or absolute. This table is called the primary region table or segment table since it is used to translate virtual addresses in the primary address space.

**Primary Subspace-Group Control (G):** Bit 54 of control register 1, when one, indicates that the address space specified by the PASCE is the base space or a subspace of a subspace group. When bit 54 is zero, the address space is not in a subspace group.

**Primary Private-Space Control (P):** If bit 55 of control register 1 is one, then (1) a one value of the common-segment bit in a translation-lookaside-buffer (TLB) representation of a segment-table entry prevents the entry and the TLB page-table copy it designates from being used when translating references to the primary address space, even with a match between the table or token origin in control register 1 and the table origin in the TLB entry, (2) low-address protection and fetch-protection override do not apply to the primary address space; and (3) a translation-specification exception is recognized if a reference to the primary address space is translated by means of a segment-table entry in storage and the common-segment bit is one in the entry. Item 2 in the above list applies even when the contents of control register 1 are a real-space designation.

**Programming Note:** With respect to item 1 in the above list when the contents of control register 1 are a real-space designation, a one value of the common-segment bit in a TLB representation of a segment-table entry prevents the entry and the TLB page-table copy it designates from being used regardless of the value of the private-space control in the real-space designation.

**Primary Storage-Alteration-Event Control (S):** When the storage-alteration-space control in control register 9 is one, bit 56 of control register 1 specifies, when one, that the primary address space is one for which storage-alteration events can occur. Bit 56 is examined when the PASCE is used to perform dynamic-address translation for a storage-operand store reference. Bit 56 is ignored when the storage-alteration-space control is zero.

**Primary Space-Switch-Event Control (X):** When bit 57 of control register 1 is one:

- A space-switch-event program interruption occurs when execution of the space-switching form of PROGRAM CALL (PC-ss), PROGRAM RETURN (PR-ss), or PROGRAM TRANSFER (PT-ss) is completed. The inter-
ruption occurs if bit 57 is one either before or after the operation.

- A space-switch-event program interruption occurs upon completion of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the address space from which instructions are fetched either to or from the home address space; that is, when instructions are fetched from the home address space either before or after the operation but not both before and after the operation.

- Condition code 3 is set by LOAD ADDRESS SPACE PARAMETERS.

**Primary Real-Space Control (R):** If bit 58 of control register 1 is zero, the register contains a region-table or segment-table designation. If bit 58 is one, the register contains a real-space designation. When bit 58 is one, a one value of the common-segment bit in a translation-lookaside-buffer (TLB) representation of a segment-table entry prevents the entry and the TLB page-table copy it designates from being used when translating references to the primary address space, even with a match between the token origin in control register 1 and the table origin in the TLB entry.

**Primary Designation-Type Control (DT):** When R is zero, the type of table designation in control register 1 is specified by bits 60 and 61 in the register, as follows:

<table>
<thead>
<tr>
<th>Bits 60 and 61</th>
<th>Designation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Region-first-table</td>
</tr>
<tr>
<td>10</td>
<td>Region-second-table</td>
</tr>
<tr>
<td>01</td>
<td>Region-third-table</td>
</tr>
<tr>
<td>00</td>
<td>Segment-table</td>
</tr>
</tbody>
</table>

When R is zero, bits 60 and 61 must be 11 binary when an attempt is made to use the PASCE to translate a virtual address in which the leftmost one bit is in bit positions 0-10 of the address. Similarly, bits 60 and 61 must be 11 or 10 binary when the leftmost one bit is in bit positions 11-21 of the address, and they must be 11, 10, or 01 binary when the leftmost one bit is in bit positions 22-32 of the address. Otherwise, an ASCE-type exception is recognized.

**Primary Region-Table or Segment-Table Length (TL):** Bits 62 and 63 of the primary region-table designation or segment-table designation in control register 1 specify the length of the primary region table or segment table in units of 4,096 bytes, thus making the length of the region table or segment table variable in multiples of 512 entries. The length of the primary region table or segment table, in units of 4,096 bytes, is one more than the TL value. The contents of the length field are used to establish whether the portion of the virtual address (RFX, RSX, RTX, or SX) to be translated by means of the table designates an entry that falls within the table.

**Primary Real-Space Token Origin:** Bits 0-51 of the primary real-space designation in control register 1, with 12 zeros appended on the right, form a 64-bit address that may be used in forming and using TLB entries that provide a virtual-equals-real translation for references to the primary address space. Although this address is used only as a token and is not used to perform a storage reference, it still must be a valid address; otherwise, an incorrect TLB entry may be used when the contents of control register 1 are used.

The following bits of control register 1 are not assigned and are ignored: bits 52, 53, and 59 if the register contains a region-table designation or segment-table designation, and bits 52, 53 and 59-63 if the register contains a real-space designation.

**Control Register 7**

Control register 7 contains the secondary address-space-control element (SASCE). The register has one of the following two formats, depending on the real-space-control bit (R) in the register:

**Secondary Region-Table or Segment-Table Designation (R=0)**

<table>
<thead>
<tr>
<th>Secondary Region-Table or Segment-Table Origin</th>
<th>G</th>
<th>P</th>
<th>S</th>
<th>R</th>
<th>DT</th>
<th>TL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>54</td>
<td>58</td>
<td>60</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

**Secondary Real-Space Designation (R=1)**

<table>
<thead>
<tr>
<th>Secondary Real-Space Token Origin</th>
<th>G</th>
<th>P</th>
<th>S</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>54</td>
<td>58</td>
<td>63</td>
</tr>
</tbody>
</table>

The secondary region-table origin, secondary segment-table origin, secondary subspace-group
control (G), secondary private-space control (P), secondary storage-alteration-event control (S), secondary real-space control (R), secondary designation-type control (DT), secondary region-table or segment-table length (TL), and secondary real-space token origin in control register 7 are defined the same as the fields in the same bit positions in control register 1, except that control register 7 applies to the secondary address space.

The following bits of control register 7 are not assigned and are ignored: bits 52, 53, 57, and 59 if the register contains a region-table designation or segment-table designation, and bits 52, 53, 57, and 59-63 if the register contains a real-space designation.

Control Register 13

Control register 13 contains the home address-space-control element (HASCE). The register has one of the following two formats, depending on the real-space-control bit (R) in the register:

**Home Region-Table or Segment-Table Designation (R=0)**

```
<table>
<thead>
<tr>
<th>Home Region-Table or Segment-Table Origin</th>
<th>P</th>
<th>S</th>
<th>X</th>
<th>R</th>
<th>DT</th>
<th>TL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>52</td>
<td>54</td>
<td>58</td>
<td>60</td>
<td>63</td>
</tr>
</tbody>
</table>
```

**Home Real-Space Designation (R=1)**

```
<table>
<thead>
<tr>
<th>Home Real-Space Token Origin</th>
<th>P</th>
<th>S</th>
<th>X</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>52</td>
<td>54</td>
<td>58</td>
</tr>
</tbody>
</table>
```

**Home Space-Switch-Event Control (X):** When bit 57 of control register 13 is one, a space-switch-event program interruption occurs upon completion of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the address space from which instructions are fetched either to or from the home address space; that is, when instructions are fetched from the home address space either before or after the operation but not both before and after the operation.

The home region-table origin, home segment-table origin, home private-space control (P), home storage-alteration-event control (S), home real-space control (R), home designation-type control (DT), home region-table or segment-table length (TL), and home real-space token origin in control register 13 are defined the same as the fields in the same bit positions in control register 1, except that control register 13 applies to the home address space.

The following bits of control register 13 are not assigned and are ignored: bits 52-54 and 59 if the register contains a region-table designation or segment-table designation, and bits 52-54 and 59-63 if the register contains a real-space designation.

Programming Notes:

1. The validity of the information loaded into a control register, including that pertaining to dynamic address translation, is not checked at the time the register is loaded. This information is checked and the program exception, if any, is indicated at the time the information is used.

2. The information pertaining to dynamic address translation is considered to be used when an instruction is executed with DAT on or when INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, LOAD REAL ADDRESS, or STORE REAL ADDRESS is executed. The information is not considered to be used when the PSW specifies translation but an I/O, external, restart, or machine-check interruption occurs before an instruction is executed, or when the PSW specifies the wait state.

Translation Tables

When the address-space-control element (ASCE) used in a translation is a region-first-table designation, the translation process consists in a five-level lookup using five tables: a region first table, a region second table, a region third table a segment table, and a page table. These tables reside in real or absolute storage. When the ASCE is a region-second-table designation, region-third-table designation, or segment-table designation, the lookups in the levels of tables above the designated level are omitted, and the higher-level tables themselves are omitted.
**Region-Table Entries**

The term “region-table entry” means a region-first-table entry, region-second-table entry, or region-third-table entry.

The entries fetched from the region first table, region second table, and region third table have the following formats. The level (first, second, or third) of the table containing an entry is identified by the table-type (TT) bits in the entry.

**Region-First-Table Entry (TT=11)**

```
<table>
<thead>
<tr>
<th>Region-Second-Table Origin</th>
<th>TF</th>
<th>I</th>
<th>TT</th>
<th>TL</th>
</tr>
</thead>
</table>
0                           | 52  | 56  | 58  | 60  | 63 |
```

**Region-Second-Table Entry (TT=10)**

```
<table>
<thead>
<tr>
<th>Region-Third-Table Origin</th>
<th>TF</th>
<th>I</th>
<th>TT</th>
<th>TL</th>
</tr>
</thead>
</table>
0                           | 52  | 56  | 58  | 60  | 63 |
```

**Region-Third-Table Entry (TT=01)**

```
<table>
<thead>
<tr>
<th>Segment-Table Origin</th>
<th>TF</th>
<th>I</th>
<th>TT</th>
<th>TL</th>
</tr>
</thead>
</table>
0                           | 52  | 56  | 58  | 60  | 63 |
```

The fields in the three levels of region-table entries are allocated as follows:

**Region-Second-Table Origin, Region-Third-Table Origin, and Segment-Table Origin:** A region-first-table entry contains a region-second-table origin. A region-second-table entry contains a region-third-table origin. A region-third-table entry contains a segment-table origin. The following description applies to each of the three origins. Bits 0-51 of the entry, with 12 zeros appended on the right, form a 64-bit address that designates the beginning of the next-lower-level table. It is unpredictable whether the address is real or absolute.

**Region-Second-Table Offset, Region-Third-Table Offset, and Segment-Table Offset (TF):** A region-first-table entry contains a region-second-table offset. A region-second-table entry contains a region-third-table offset. A region-third-table entry contains a segment-table offset. The following description applies to each of the three offsets. Bits 56 and 57 of the entry specify the length of a portion of the next-lower-level table that is missing at the beginning of the table, that is, the bits specify the location of the first entry actually existing in the next-lower-level table. The bits specify the length of the missing portion in units of 4,096 bytes, thus making the length of the missing portion variable in multiples of 512 entries. The length of the missing portion, in units of 4,096 bytes, is equal to the TF value. The contents of the offset field, in conjunction with the length field, bits 62 and 63, are used to establish whether the portion of the virtual address (RSX, RTX, or SX) to be translated by means of the next-lower-level table designates an entry that actually exists in the table.

**Region-Invalid Bit (I):** Bit 58 in a region-first-table entry or region-second-table entry controls whether the set of regions associated with the entry is available. Bit 58 in a region-third-table entry controls whether the single region associated with the entry is available. When bit 58 is zero, address translation proceeds by using the region-table entry. When the bit is one, the entry cannot be used for translation.

**Table-Type Bits (TT):** Bits 60 and 61 of the region-first-table entry, region-second-table entry, and region-third-table entry identify the level of the table containing the entry, as follows:

<table>
<thead>
<tr>
<th>Bits 60 and 61</th>
<th>Region-Table Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>First</td>
</tr>
<tr>
<td>10</td>
<td>Second</td>
</tr>
<tr>
<td>01</td>
<td>Third</td>
</tr>
</tbody>
</table>

Bits 60 and 61 must identify the correct table level, considering the type of table designation that is the address-space-control element being used in the translation and the number of table levels that have so far been used; otherwise, a translation-specification exception is recognized.

**Region-Second-Table Length, Region-Third-Table Length, and Segment-Table Length (TL):** A region-first-table entry contains a region-second-table length. A region-second-table entry contains a region-third-table length. A region-third-table entry contains a segment-table length. The following description applies to each of the three lengths. Bits 62 and 63 of the entry specify the length of the next-lower-level table in units of 4,096 bytes, thus making the length of the table variable in multiples of 512 entries. The length of the next-lower-level table, in units of 4,096 bytes, is one more than the TL value. The contents of
the length field, in conjunction with the offset field, bits 56 and 57, are used to establish whether the portion of the virtual address (RSX, RTX, or SX) to be translated by means of the next-lower-level table designates an entry that actually exists in the table.

**Segment-Table Entries**
The entry fetched from the segment table has the following format:

```
Segment-Table Entry (TT=00)
```

<table>
<thead>
<tr>
<th>Page-Table Origin</th>
<th>P</th>
<th>I</th>
<th>C</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>53</td>
<td>55</td>
<td>58</td>
<td>60</td>
</tr>
</tbody>
</table>

The fields in the segment-table entry are allocated as follows:

**Page-Table Origin:** Bits 0-52, with 11 zeros appended on the right, form a 64-bit address that designates the beginning of a page table. It is unpredictable whether the address is real or absolute.

**Page-Protection Bit (P):** Bit 54 is treated as being ORed with the page-protection bit in each entry in the page table designated by this segment-table entry. Thus, when the bit is one, page protection applies to the entire segment specified by the segment-table entry.

**Segment-Invalid Bit (I):** Bit 58 controls whether the segment associated with the segment-table entry is available. When the bit is zero, address translation proceeds by using the segment-table entry. When the bit is one, the segment-table entry cannot be used for translation.

**Common-Segment Bit (C):** Bit 59 controls the use of the translation-lookaside-buffer (TLB) copies of the segment-table entry and of the page table which it designates. A zero identifies a private segment; in this case, the segment-table entry and the page table it designates may be used only in association with the segment-table origin that designates the segment table in which the segment-table entry resides. A one identifies a common segment; in this case, the segment-table entry and the page table it designates may continue to be used for translating addresses corresponding to the segment index, even though a different segment table is specified. However, TLB copies of the segment-table entry and page table for a common segment are not usable if the private-space control, bit 55, is one in the address-space-control element used in the translation or if that address-space-control element is a real-space designation. The common-segment bit must be zero if the segment-table entry is fetched from storage during a translation when the private-space control is one in the address-space-control element being used; otherwise, a translation-specification exception is recognized.

**Table-Type Bits (TT):** Bits 60 and 61 of the segment-table entry are 00 binary to identify the level of the table containing the entry. The meanings of all possible values of bits 60 and 61 in a region-table entry or segment-table entry are as follows:

<table>
<thead>
<tr>
<th>Bits 60 and 61</th>
<th>Table Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Region-First</td>
</tr>
<tr>
<td>10</td>
<td>Region-Second</td>
</tr>
<tr>
<td>01</td>
<td>Region-Third</td>
</tr>
<tr>
<td>00</td>
<td>Segment</td>
</tr>
</tbody>
</table>

Bits 60 and 61 must identify the correct table level, considering the type of table designation that is the address-space-control element being used in the translation and the number of table levels that have so far been used; otherwise, a translation-specification exception is recognized.

Bits 53, 55-57, 62, and 63 of the segment-table entry are reserved for possible future extensions.

**Programming Note:** When the common-segment (C) bit is one in a valid segment-table entry fetched from storage during dynamic-address translation for a non-private address space, that is, for an address space in which the private-space control is zero in the address-space-control element being used, then the following conditions apply:

1. The segment-table entries corresponding to the same virtual address in all other non-private address spaces must be identical.

   If the program alters such a segment-table entry in one non-private address space, then it must also (a) identically alter all other segment-table entries corresponding to the same virtual address in all other non-private address spaces, and (b) ensure that the affected entries are cleared from the TLBs of
all CPUs in the configuration. Further information on clearing TLB entries may be found in “Modification of Translation Tables” on page 3-52.

2. The program must ensure that segment-table entries in which the C bit is set to one, and any region-table entries used by the DAT process to locate such segment-table entries, are consistent across all address spaces. That is, if the DAT process can successfully locate a segment-table entry in which the C bit is one in one non-private address space, then there must be no exception condition that prevents DAT from locating the segment-table entry corresponding to the same virtual address in any other non-private address spaces.

If the program alters such a segment-table entry or region-table entry in one non-private address space, then it must also (a) perform consistent alteration to all corresponding table entries in all other non-private address spaces, and (b) ensure that the affected entries are cleared from the TLBs of all CPUs in the configuration.

3. If the program fails to maintain consistent DAT table entries as described above, results are unpredictable and may include the presentation of a delayed-access-exception machine check, as described in “Delayed Access Exception” on page 11-18.

Page-Table Entries
The entry fetched from the page table entry has the following format:

```
+----------+-------+-----+-----+
| PFRA     | I     | P   +-----|
| Page-Frame Real Address | 0 | 1 | 0 |
+----------+-------+-----+-----+
```

The fields in the page-table entry are allocated as follows:

**Page-Frame Real Address (PFRA):** Bits 0-51 provide the leftmost bits of a real storage address. When these bits are concatenated with the 12-bit byte-index field of the virtual address on the right, a 64-bit real address is obtained.

**Page-Invalid Bit (I):** Bit 53 controls whether the page associated with the page-table entry is available. When the bit is zero, address translation proceeds by using the page-table entry. When the bit is one, the page-table entry cannot be used for translation.

**Page-Protection Bit (P):** Bit 54 controls whether store accesses can be made in the page. This protection mechanism is in addition to the key-controlled-protection and low-address-protection mechanisms. The bit has no effect on fetch accesses. If the bit is zero, stores are permitted to the page, subject to the page-protection bit in the segment-table entry used in the translation and to the other protection mechanisms. If the bit is one, stores are disallowed. An attempt to store when the page-protection bit is one causes a protection exception to be recognized. The page-protection bit in the segment-table entry is treated as being ORed with bit 54 when determining whether page protection applies to the page.

Bit positions 52 and 55 of the entry must contain zeros; otherwise, a translation-specification exception is recognized as part of the execution of an instruction using that entry for address translation. Bit positions 56-63 are not assigned and are ignored.

Translation Process
This section describes the translation process as it is performed implicitly before a virtual address is used to access main storage. Explicit translation, which is the process of translating the operand address of LOAD REAL ADDRESS, STORE REAL ADDRESS, and TEST PROTECTION, is the same, except that, for LOAD REAL ADDRESS and TEST PROTECTION, region-first-translation, region-second-translation, region-third-translation, segment-translation, and page-translation exceptions are not recognized; such conditions are instead indicated by the condition code. Translation of the operand address of LOAD REAL ADDRESS and STORE REAL ADDRESS also differs in that the CPU may be in the real mode.

Translation of a virtual address is controlled by the DAT-mode bit and address-space-control bits in the PSW and by the address-space-control elements (ASCEs) in control registers 1, 7, and 13 and as specified by the access registers. When the ASCE used in a translation is a region-first-table designation, the translation is performed by means of a region first table, region second table,
region third table, segment table, and page table, all of which reside in real or absolute storage. When the ASCE is a lower-level type of table designation (region-second-table designation, region-third-table designation, or segment-table designation) the translation is performed by means of only the table levels beginning with the designated level, and the virtual-address bits that would, if nonzero, require use of a higher level or levels of table must be all zeros; otherwise, an ASCE-type exception is recognized. When the ASCE is a real-space designation, the virtual address is treated as a real address, and table entries in real or absolute storage are not used.

The address-space-control element (ASCE) used for a particular address translation is called the effective ASCE. Accordingly, when a primary virtual address is translated, the contents of control register 1 are used as the effective ASCE. Similarly, for a secondary virtual address, the contents of control register 7 are used; for an AR-specified virtual address, the ASCE specified by the access register is used; and for a home virtual address, the contents of control register 13 are used.

When the real-space control in the effective ASCE is zero, the designation-type control in the ASCE specifies the table-designation type of the ASCE: region-first-table designation, region-second-table designation, region-third-table designation, or segment-table designation. The corresponding portion of the virtual address (region first index, region second index, region third index, or segment index) is checked against the table-length field in the designation, and it is added to the origin in the designation to select an entry in the designated table. If the selected entry is outside its table, as determined by the table-offset and table-length fields in the current table entry, or if the I bit is one in the selected entry, a region-second-translation, region-third-translation, or segment-translation exception is recognized, depending on the level of the next table. If the table-type bits in the selected entry do not indicate the expected table level, a translation-specification exception is recognized.

Processing of portions of the virtual address by means of successive table levels continues until a segment-table entry has been selected. This entry designates the page table to be used. The segment-table entry contains a page-protection bit that applies to all pages in the specified segment.

The page-index portion of the virtual address is added to the page-table origin in the segment-table entry to select an entry in the page table. If the I bit is one in the page-table entry, a page-translation exception is recognized. The page-table entry contains the leftmost bits of the real address that represents the translation of the virtual address, and it contains a page-protection bit that applies only to the page specified by the page-table entry.

The byte-index field of the virtual address is used unchanged as the rightmost bit positions of the real address.

In order to eliminate the delay associated with references to translation tables in real or absolute storage, the information fetched from the tables normally is also placed in a special buffer, the translation-lookaside buffer (TLB), and subsequent translations involving the same table entries may be performed by using the information recorded in the TLB. The TLB may also record virtual-equals-real translations related to a real-space designation. The operation of the TLB is described in "Translation-Lookaside Buffer” on page 3-50.

Whenever access to real or absolute storage is made during the address-translation process for the purpose of fetching an entry from a region
table, segment table, or page table, key-controlled protection does not apply.

The translation process, including the effect of the TLB, is shown graphically in Figure 3-10.

Control Reg. 1, 7, or 13

PASCE, SASCE, or HASCE

ASN-Second Table Entry

AR-Specified ASCE

Virtual Address

RFX RSX RTX SX PX BX

(x8) (x8) (x8) (x8)

1

Effective ASCE

TO R DT TL TO and virtual address → 5

R=1 ?

Yes ASCE is RSD

5

No

If DT=11 +

Region First Table

R/A

RSTO TF I TT TL

A in Part 2

If DT=10 +

Region Second Table

R/A

RTTO TF I TT TL

B in Part 2

If DT=01 +

Region Third Table

R/A

STO TF I TT TL

C in Part 2

If DT=00 +

Segment Table

R/A

PTO P I C TT

D in Part 3

R/A: Address is real or absolute.

Figure 3-10 (Part 1 of 3). Translation Process
Region-First-Table Entry

A

RSTO TF I TT TL

(x4096)

+ 

Region Second Table

3

R/A RTTO TF I TT TL B

Region-Second-Table Entry

B

RTTO TF I TT TL

(x4096)

+ 

Region Third Table

3

R/A STO TF I TT TL C

Region-Third-Table Entry

C

STO TF I TT TL

(x4096)

+ 

Segment Table

3

R/A PTO P I C TT D in Part 3

R/A: Address is real or absolute.

Figure 3-10 (Part 2 of 3). Translation Process
Control register 1 provides the primary address-space-control element (ASCE) for translation of a primary virtual address, control register 7 provides the secondary ASCE for translation of a secondary virtual address, and control register 13 provides the home ASCE for translation of a home virtual address. An ASN-second-table entry provides an AR-specified (access-register-specified) ASCE for translation of an AR-specified virtual address.

The portion of the virtual address to the left of the index selected by DT must be zero; otherwise, an ASCE-type exception is recognized. Bits 0 and 1 of the index must be less than or equal to TL in the ASCE, and I in the selected table entry must be zero; otherwise, a region-first-translation, region-second-translation, region-third-translation or segment-translation exception is recognized, depending on the table level selected by DT. TT in the selected table entry must equal DT; otherwise, a translation-specification exception is recognized.

Bits 0 and 1 of the next index must be equal to or greater than TF, and less than or equal to TL, in the current table entry, and I in the next selected table entry must be zero; otherwise, a region-second-translation, region-third-translation, or segment-translation exception is recognized, depending on the table level of the next selected entry. TT in the next selected entry must be one less than TT in the current entry; otherwise, a translation-specification exception is recognized.

I in the page-table entry must be zero; otherwise, a page-translation exception is recognized. Bits 52 and 55 in the page-table entry must be zero; otherwise, a translation-specification exception is recognized.

Information, which may include portions of the virtual address and the table origin or real-space token origin in the effective ASCE, is used to search the TLB.

If a match exists, the page-frame real address from the TLB is used in forming the real address. If no match exists and the effective ASCE is a table designation, table entries in real or absolute storage are fetched. The resulting fetched entries are used to translate the address and, in conjunction with the search information, may be used to form entries in the TLB. If the effective ASCE is a real-space designation, a TLB entry that translates the virtual address to the equal real address may be formed.

Figure 3-10 (Part 3 of 3). Translation Process
Inspection of Real-Space Control
When the effective address-space-control element (ASCE) contains a real-space control, bit 58, having the value zero, the ASCE is a region-table or segment-table designation. When the real-space control is one, the ASCE is a real-space designation.

Inspection of Designation-Type Control
When the real-space control is zero, the designation-type control, bits 60 and 61 of the effective address-space-control element (ASCE), specifies the table-designation type of the ASCE. Depending on the type, some number of leftmost bits of the virtual address being translated must be zeros; otherwise, an ASCE-type exception is recognized. For each possible value of bits 60 and 61, the table-designation type and the virtual-address bits required to be zeros are as follows:

<table>
<thead>
<tr>
<th>Bits 60 and 61</th>
<th>Designation Type</th>
<th>Virtual-Address Bits Required to Be Zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Region-first-table</td>
<td>None</td>
</tr>
<tr>
<td>10</td>
<td>Region-second-table</td>
<td>0-10</td>
</tr>
<tr>
<td>01</td>
<td>Region-third-table</td>
<td>0-21</td>
</tr>
<tr>
<td>00</td>
<td>Segment-table</td>
<td>0-32</td>
</tr>
</tbody>
</table>

Lookup in a Table Designated by an Address-Space-Control Element
The designation-type control, bits 60 and 61 of the effective address-space-control element (ASCE), specifies both the table-designation type of the ASCE and the portion of the virtual address that is to be translated by means of the designated table, as follows:

When bits 60 and 61 have the value 11 binary, the region-first-index portion of the virtual address, in conjunction with the region-first-table origin contained in the ASCE, is used to select an entry from the region first table.

The 64-bit address of the region-first-table entry in real or absolute storage is obtained by appending 12 zeros to the right of bits 0-51 of the region-first-table designation and adding the region first index with three rightmost and 50 leftmost zeros appended. When a carry out of bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{64} - 1$ to zero. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode. When forming the address of a region-first-, region-second-, region-third-, or segment-table entry, it is unpredictable whether prefixing, if any, is applied to the respective table origin contained in the ASCE before the addition of the table index value, or prefixing is applied to the table-entry address that is formed by the addition of the table origin and table index value.

As part of the region-first-table-lookup process, bits 0 and 1 of the virtual address (which are bits 0 and 1 of the region first index) are compared against the table length, bits 62 and 63 of the region-first-table designation, to establish whether the addressed entry is within the region first table. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-first-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a
region-first-table entry in the translation-lookaside buffer is used in the translation.

All eight bytes of the region-first-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the region-first-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

Bit 58 of the entry fetched from the region first table specifies whether the corresponding set of regions is available. This bit is inspected, and, if it is one, a region-first-translation exception is recognized.

A translation-specification exception is recognized if the table-type bits, bits 60 and 61, in the region-first-table entry do not have the same value as bits 60 and 61 of the ASCE.

When no exceptions are recognized in the process of region-first-table lookup, the entry fetched from the region first table designates the beginning and specifies the offset and length of the corresponding region second table.

When bits 60 and 61 of the ASCE have the value 01 binary, the region-third-index portion of the virtual address, in conjunction with the region-third-table origin contained in the ASCE, is used to select an entry from the region third table. Bits 22 and 23 of the virtual address (which are bits 0 and 1 of the region third index) are compared against the table length in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-third-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a region-third-table entry in the translation-lookaside buffer is used in the translation. The region-third-table-lookup process is otherwise the same as the region-first-table-lookup process, including the checking of the table-type bits in the region-third-table entry, except that a region-third-translation exception is recognized if bit 58 is one in the region-third-table entry. When no exceptions are recognized, the entry fetched from the region third table designates the beginning and specifies the offset and length of the corresponding region third table.

When bits 60 and 61 of the ASCE have the value 00 binary, the segment-index portion of the virtual address, in conjunction with the segment-table origin contained in the ASCE, is used to select an entry from the segment table. Bits 33 and 34 of the virtual address (which are bits 0 and 1 of the segment index) are compared against the table length in the ASCE. If the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a segment-translation exception is recognized. The comparison against the table length may be omitted if the equivalent of a segment-table entry in the translation-lookaside buffer is used in the translation. A translation-specification exception is recognized if (1) the private-space control, bit 55, in the ASCE is one and (2) the common-segment bit, bit 59, in the entry fetched from the segment table is one. The segment-table-lookup process is otherwise the same as the region-first-table-lookup process, including the checking of the table-type bits in the segment-table entry, except that a segment-translation exception is recognized if bit 58 is one in the segment-table entry. When no exceptions are recognized, the entry fetched from the segment table designates the beginning of the corresponding page table.
Lookup in a Table Designated by a Region-Table Entry

When the effective address-space-control element (ASCE) is a region-table designation, a region-table entry is selected as described in the preceding section. Then the contents of the selected entry and the next index portion of the virtual address are used to select an entry in the next-lower-level table, which may be another region table or a segment table.

When the table entry selected by means of the ASCE is a region-first-table entry, the region-second-index portion of the virtual address, in conjunction with the region-second-table origin contained in the region-first-table entry, is used to select an entry from the region second table.

The 64-bit address of the region-second-table entry in real or absolute storage is obtained by appending 12 zeros to the right of bits 0-51 of the region-first-table entry and adding the region second index with three rightmost and 50 leftmost zeros appended. When a carry out of bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{64} - 1$ to zero. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode. When forming the address of a region-second-, region-third-, or segment-table entry, it is unpredictable whether prefixing, if any, is applied to the respective table origin contained in the higher-level table entry before the addition of the table index value, or prefixing is applied to the table-entry address that is formed by the addition of the table origin and table index value.

As part of the region-second-table-lookup process, bits 11 and 12 of the virtual address (which are bits 0 and 1 of the region second index) are compared against the table offset, bits 56 and 57 of the region-first-table entry, and against the table length, bits 62 and 63 of the region-first-table entry, to establish whether the addressed entry is within the region second table. If the value in the table-offset field is greater than the value in the corresponding bit positions of the virtual address, or if the value in the table-length field is less than the value in the corresponding bit positions of the virtual address, a region-second-translation exception is recognized.

All eight bytes of the region-second-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the region-second-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

Bit 58 of the entry fetched from the region second table specifies whether the corresponding set of regions is available. This bit is inspected, and, if it is one, a region-second-translation exception is recognized.

A translation-specification exception is recognized if the table-type bits, bits 60 and 61, in the region-second-table entry do not have a value that is one less than the value of those bits in the next-higher-level table.

When no exceptions are recognized in the process of region-second-table lookup, the entry fetched from the region second table designates the beginning and specifies the offset and length of the corresponding region third table.

When the table entry selected by means of the ASCE is a region-second-table entry, or if a region-second-table entry has been selected by means of the contents of a region-first-table entry, the region-third-index portion of the virtual address, in conjunction with the region-third-table origin contained in the region-second-table entry, is used to select an entry from the region third table. Bits 22 and 23 of the virtual address (which are bits 0 and 1 of the region third index) are compared against the table offset and table length in the region-second-table entry. A region-third-translation exception is recognized if the table offset is greater than bits 22 and 23 or if the table length is less than bits 22 and 23. The region-third-table-lookup process is otherwise the same as the region-second-table-lookup process, including the checking of the table-type bits in the region-third-table entry, except that a region-third-translation exception is recognized if bit 58 is one in the region-third-table entry. When no exceptions are recognized, the entry fetched from the region third table designates the beginning and specifies the offset and length of the corresponding segment table.
When the table entry selected by means of the ASCE is a region-third-table entry, or if a region-third-table entry has been selected by means of the contents of a region-second-table entry, the segment-index portion of the virtual address, in conjunction with the segment-table origin contained in the region-third-table entry, is used to select an entry from the segment table. Bits 33 and 34 of the virtual address (which are bits 0 and 1 of the segment index) are compared against the table offset and table length in the region-third-table entry. A segment-translation exception is recognized if the table offset is greater than bits 33 and 34 or if the table length is less than bits 33 and 34. A translation-specification exception is recognized if (1) the private-space control, bit 55, in the ASCE is one and (2) the common-segment bit, bit 59, in the entry fetched from the segment table is one. The segment-table-lookup process is otherwise the same as the region-second-table-lookup process, including the checking of the table-type bits in the segment-table entry, except that a segment-translation exception is recognized if bit 58 is one in the segment-table entry. When no exceptions are recognized, the entry fetched from the segment table designates the beginning of the corresponding page table.

**Page-Table Lookup**
The page-index portion of the virtual address, in conjunction with the page-table origin contained in the segment-table entry, is used to select an entry from the page table.

The 64-bit address of the page-table entry in real or absolute storage is obtained by appending 11 zeros to the right of the page-table origin and adding the page index, with three rightmost and 53 leftmost zeros appended. A carry out of bit position 0 cannot occur. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All eight bytes of the page-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address generated for fetching the page-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the unit of operation is suppressed.

The entry fetched from the page table indicates the availability of the page and contains the leftmost bits of the page-frame real address. The page-invalid bit, bit 53, is inspected to establish whether the corresponding page is available. If this bit is one, a page-translation exception is recognized. If bit position 52 or 55 contains a one, a translation-specification exception is recognized. If the page-protection bit, bit 54, is one either in the segment-table entry used in the translation or in the page-table entry, and the storage reference for which the translation is being performed is a store, a protection exception is recognized.

**Formation of the Real Address**
When the effective address-space-control element (ASCE) is a region-table designation or a segment-table designation and no exceptions in the translation process are encountered, the page-frame real address is obtained from the page-table entry. When the ASCE is a real-space designation, bits 0-51 of the virtual address are used as a page-frame real address. In either case, the page-frame real address and the byte-index portion of the virtual address are concatenated, with the page-frame real address forming the leftmost part. The result is the real storage address which corresponds to the virtual address. All 64 bits of the address are used, regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

**Recognition of Exceptions during Translation**
Invalid addresses and invalid formats can cause exceptions to be recognized during the translation process. Exceptions are recognized when information contained in table entries is used for translation and is found to be incorrect.

The information pertaining to DAT is considered to be used when an instruction is executed with DAT on or when INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, LOAD REAL ADDRESS, or STORE REAL ADDRESS is executed. The information is not considered to be used when the PSW specifies DAT on but an I/O, external, restart, or machine-check interruption occurs before an instruction is executed, or when the PSW specifies the wait state. Only that information required in order to translate a virtual address is considered to be in use during the
translation of that address, and, in particular, addressing exceptions that would be caused by the use of an address-space-control element are not recognized when that address-space-control element is not the one actually used in the translation.

A list of translation exceptions, with the action taken for each exception and the priority in which the exceptions are recognized when more than one is applicable, is provided in "Recognition of Access Exceptions" on page 6-36.

Translation-Lookaside Buffer

To enhance performance, the dynamic-address-translation mechanism normally is implemented such that some of the information specified in the region tables, segment tables, and page tables is maintained in a special buffer, referred to as the translation-lookaside buffer (TLB). The CPU necessarily refers to a DAT-table entry in real or absolute storage only for the initial access to that entry. This information may be placed in the TLB, and subsequent translations may be performed by using the information in the TLB. For consistency of operation, the virtual-equals-real translation specified by a real-space designation also may be performed by using information in the TLB. The presence of the TLB affects the translation process to the extent that (1) a modification of the contents of a table entry in real or absolute storage does not necessarily have an immediate effect, if any, on the translation, (2) a region-first-table origin, region-second-table origin, region-third-table origin, segment-table origin, or real-space token origin in an address-space-control element (ASCE) may select a TLB entry that was formed by means of an ASCE containing an origin of the same value even when the two origins are of different types, and (3) the comparison against the table length in an address-space-control element may be omitted if a TLB equivalent of the designated table entry is used. In a multiple-CPU configuration, each CPU has its own TLB.

Entries within the TLB are not explicitly addressable by the program.

Information is not necessarily retained in the TLB under all conditions for which such retention is permissible. Furthermore, information in the TLB may be cleared under conditions additional to those for which clearing is mandatory.

TLB Structure

The description of the logical structure of the TLB covers the implementation by all systems operating as defined by z/Architecture. The TLB entries are considered as being of three types: TLB combined region-and-segment-table entries, TLB page-table entries, and TLB real-space entries. A TLB combined region-and-segment-table entry or TLB page-table entry is considered as containing within it both the information obtained from the table entry or entries in real or absolute storage and the attributes used to fetch this information from storage. A TLB real-space entry is considered as containing a page-frame real address and the real-space token origin and region, segment, and page indexes used to form the entry. The token origin in a TLB real-space entry is indistinguishable from the table origin in a TLB combined region-and-segment-table entry.

Note: The following sections describe the conditions under which information may be placed in the TLB, the conditions under which information from the TLB may be used for address translation, and how changes to the translation tables affect the translation process.

Formation of TLB Entries

The formation of TLB combined region-and-segment-table entries and TLB page-table entries from table entries in real or absolute storage, and the effect of any manipulation of the contents of table entries in storage by the program, depend on whether the entries in storage are attached to a particular CPU and on whether the entries are valid.

The attached state of a table entry denotes that the CPU to which it is attached can attempt to use the table entry for implicit address translation, except that a table entry for the primary or home address space may be attached even when the CPU does not fetch from either of those spaces. A table entry may be attached to more than one CPU at a time.

The valid state of a table entry denotes that the region set, region, segment, or page associated with the table entry is available. An entry is valid
when the region-invalid, segment-invalid, or page-invalid bit in the entry is zero.

The region-table entries, if any, and the segment-table entry used to form a TLB combined region-and-segment-table entry are called a translation path. A translation path may be placed in the TLB as a combined region-and-segment-table entry whenever all entries in the path are attached and valid and would not cause a translation-specification exception if used for translation. Similarly, a page-table entry may be placed in the TLB whenever the entry is attached and valid and would not cause a translation-specification exception if used for translation.

The highest-level table entry in a translation path is attached when it is within a table designated by an attaching address-space-control element (ASCE). “Within a table” means as determined by the origin and length fields in the ASCE. An ASCE is an attaching ASCE when all of the following conditions are met:

1. The current PSW specifies DAT on.
2. The current PSW contains no errors that would cause an early specification exception to be recognized.
3. The ASCE meets the requirements in a, b, c, or d below.
   a. The ASCE is the primary ASCE in control register 1.
   b. The ASCE is the secondary ASCE in control register 7, and either of the following requirements is met:
      • The CPU is in the secondary-space mode or access-register mode.
      • The CPU is in the primary-space mode, and the secondary-space control, bit 37 of control register 0, is one.
   c. The ASCE is in either an attached and valid ASN-second-table entry (ASTE) or a usable ALB ASTE, and the CPU is in the access-register mode. See “ART-Lookaside Buffer” on page 5-62 for the meaning of the terminology used here.
   d. The ASCE is the home ASCE in control register 13.

Each of the remaining table entries in a translation path is attached when the next-higher-level entry is attached and valid and would not cause a translation-specification exception if used for translation and the subject entry is within the table designated by the next-higher-level entry. “Within the table” means as determined by the origin, offset, and length fields in the next-higher-level entry.

A page-table entry is attached when it is within the page table designated by either an attached and valid segment-table entry that would not cause a translation-specification exception if used for translation or a usable TLB combined region-and-segment-table entry. A usable TLB combined region-and-segment-table entry is explained in the next section.

A region-table entry or segment-table entry causes a translation-specification exception if the table-type bits, bits 60 and 61, in the entry are inconsistent with the level at which the entry would be encountered when using the translation path in the translation process. A segment-table entry also causes a translation-specification exception if the private-space-control bit is one in the address-space-control element used to select it and the common-segment bit is one in the entry. A page-table entry causes a translation-specification exception if bit 52 or 55 in the entry is one.

A TLB real-space entry may be formed whenever an attaching real-space designation exists. The entry is formed using the real-space token origin in the designation and any value of bits 0-51 of a virtual address.

Use of TLB Entries
The usable state of a TLB entry denotes that the CPU can attempt to use the TLB entry for implicit address translation. A usable TLB entry attaches the next-lower-level table, if any, and may be usable for a particular instance of implicit address translation.

A TLB combined region-and-segment-table entry is in the usable state when all of the following conditions are met:

1. The current PSW specifies DAT on.
2. The current PSW contains no errors that would cause an early specification exception to be recognized.
3. The TLB combined region-and-segment-table entry meets either of the following requirements:
   a. The common-segment bit is one in the TLB entry.
   b. The table-origin (TO) field in the TLB entry matches the table- or token-origin field in an attaching address-space-control element.

A TLB combined region-and-segment-table entry may be used for a particular instance of implicit address translation only when the entry is in the usable state, either the common-segment bit is one in the TLB entry or the table-origin (TO) field in the TLB entry matches the table- or token-origin field in the address-space-control element being used in the translation, and the region-index and segment-index fields in the TLB entry match those of the virtual address being translated. However, the TLB combined region-and-segment-table entry is not used if the common-segment bit is one in the entry and either the private-space-control bit is one in the address-space-control element being used in the translation or that address-space-control element is a real-space designation. In both these cases, the TLB entry is not used even if the table-origin field in the entry and the table- or token-origin field in the address-space-control element match.

A TLB page-table entry may be used for a particular instance of implicit address translation only when the page-table-origin field in the entry matches the page-table-origin field in the segment-table entry or TLB combined region-and-segment-table entry being used in the translation and the page-index field in the TLB page-table entry matches the page index of the virtual address being translated.

A TLB real-space entry may be used for implicit address translation only when the token-origin field in the TLB entry matches the table- or token-origin field in the address-space-control element being used in the translation and the region-index, segment-index, and page-index fields in the TLB entry match those of the virtual address being translated.

The operand address of LOAD REAL ADDRESS may be translated with the use of the TLB contents whether DAT is on or off, but TLB entries still are formed only if DAT is on.

Programming Notes:
1. Although contents of a table entry may be copied into the TLB only when the table entry is both attached and valid, the copy may remain in the TLB even when the table entry itself is no longer attached or valid.
2. No contents can be copied into the TLB when DAT is off because the table entries at this time are not attached. In particular, translation of the operand address of LOAD REAL ADDRESS with DAT off does not cause entries to be placed in the TLB. Conversely, when DAT is on, information may be copied into the TLB from all translation-table entries that could be used for address translation, given the current translation parameters, the setting of the address-space-control bits, and the contents of the access registers. The loading of the TLB does not depend on whether the entry is used for translation as part of the execution of the current instruction, and such loading can occur when the CPU is in the wait state.
3. More than one copy of contents of a table entry may exist in the TLB. For example, some implementations may cause a copy of contents of a valid table entry to be placed in the TLB for the table origin in each address-space-control element by which the entry becomes attached.

Modification of Translation Tables
When an attached and invalid table entry is made valid and no entry usable for translation of the associated virtual address is in the TLB, the change takes effect no later than the end of the current unit of operation. Similarly, when an unattached and valid table entry is made attached and no usable entry for the associated virtual address is in the TLB, the change takes effect no later than the end of the current unit of operation.

When a valid and attached table entry is changed, and when, before the TLB is cleared of entries that qualify for substitution for that entry, an attempt is made to refer to storage by using a virtual address requiring that entry for translation, unpredictable results may occur, to the following extent. The use of the new value may begin between instructions or during the execution of an instruction, including the instruction that caused
the change. Moreover, until the TLB is cleared of entries that qualify for substitution for that entry, the TLB may contain both the old and the new values, and it is unpredictable whether the old or new value is selected for a particular access. If both old and new values of a translation path are present in the TLB, a page-table entry may be fetched by using one value and placed in the TLB associated with the other value. If the new value of the path is a value that would cause an exception, the exception may or may not cause an interruption to occur. If an interruption does occur, the result fields of the instruction may be changed even though the exception would normally cause suppression or nullification.

Entries are cleared from the TLB in accordance with the following rules:

1. All entries are cleared from the TLB by the execution of PURGE TLB or SET PREFIX and by CPU reset.
2. All entries may be cleared from all TLBs in the configuration by the execution of COMPARE AND SWAP AND PURGE by any of the CPUs in the configuration, depending on a bit in a general register used by the instruction.
3. Selected entries are cleared from all TLBs in the configuration by the execution of INVALIDATE DAT TABLE ENTRY or INVALIDATE PAGE TABLE ENTRY by any of the CPUs in the configuration.
4. Some or all TLB entries may be cleared at times other than those required by the preceding rules.

Programming Notes:

1. Entries in the TLB may continue to be used for translation after the table entries from which they have been formed have become unattached or invalid. These TLB entries are not necessarily removed unless explicitly cleared from the TLB. A change made to an attached and valid entry or a change made to a table entry that causes the entry to become attached and valid is reflected in the translation process for the next instruction, or earlier than the next instruction, unless a TLB entry qualifies for substitution for that table entry. However, a change made to a table entry that causes the entry to become unattached or invalid is not necessarily reflected in the translation process until the TLB is cleared of entries that qualify for substitution for that table entry.

2. Exceptions associated with dynamic address translation may be established by a pretest for operand accessibility that is performed as part of the initiation of instruction execution. Consequently, a region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception may be indicated when a table entry is invalid at the start of execution even if the instruction would have validated the table entry it uses and the table entry would have appeared valid if the instruction was considered to process the operands one byte at a time.

3. A change made to an attached table entry, except to set the I bit to zero or to alter the rightmost byte of a page-table entry, may produce unpredictable results if that entry is used for translation before the TLB is cleared of all copies of contents of that entry. The use of the new value may begin between instructions or during the execution of an instruction, including the instruction that caused the change. When an instruction, such as MOVE (MVC), makes a change to an attached table entry, including a change that makes the entry invalid, and subsequently uses the entry for translation, a changed entry is being used without a prior clearing of the entry from the TLB, and the associated unpredictability of result values and of exception recognition applies.

Manipulation of attached table entries may cause spurious table-entry values to be recorded in a TLB. For example, if changes are made piecemeal, modification of a valid attached entry may cause a partially updated entry to be recorded, or, if an intermediate value is introduced in the process of the change, a supposedly invalid entry may temporarily appear valid and may be recorded in the TLB. Such an intermediate value may be introduced if the change is made by an I/O operation that is retried, or if an intermediate value is introduced during the execution of a single instruction.

As another example, if a segment-table entry is changed to designate a different page table and used without clearing the TLB, the new
page-table entries may be fetched and associated with the old page-table origin. In such a case, execution of INVALIDATE PAGE TABLE ENTRY designating the new page-table origin will not necessarily clear the page-table entries fetched from the new page table.

4. To facilitate the manipulation of page tables, the INVALIDATE PAGE TABLE ENTRY instruction is provided. This instruction sets the I bit in a page-table entry to one and clears all TLBs in the configuration of entries formed from that table entry.

INVALIDATE PAGE TABLE ENTRY is useful for setting the I bit to one in a page-table entry and causing TLB copies of the entry to be cleared from the TLB of each CPU in the configuration. The following aspects of the TLB operation should be considered when using INVALIDATE PAGE TABLE ENTRY. (See also the programming notes for INVALIDATE PAGE TABLE ENTRY.)

a. INVALIDATE PAGE TABLE ENTRY should be executed before making any change to a page-table entry other than changing the rightmost byte; otherwise, the selective-clearing portion of INVALIDATE PAGE TABLE ENTRY may not clear the TLB copies of the entry.

b. Invalidation of all the page-table entries within a page table by means of INVALIDATE PAGE TABLE ENTRY does not necessarily clear the TLB of any combined region-and-segment-table entry designating the page table. When it is desired to invalidate and clear the TLB of a combined region-and-segment-table entry, the rules in note 5 below must be followed.

c. When a large number of page-table entries are to be invalidated at a single time, the overhead involved in using COMPARE AND SWAP AND PURGE (one that purges the TLB), INVALIDATE DAT TABLE ENTRY, or PURGE TLB and in following the rules in note 5 below may be less than in issuing INVALIDATE PAGE TABLE ENTRY for each page-table entry.

5. Manipulation of table entries should be in accordance with the following rules. If these rules are complied with, translation is performed as if the table entries from real or absolute storage were always used in the translation process.

a. A valid table entry must not be changed while it is attached to any CPU and may be used for translation by that CPU except to (1) invalidate the entry by using INVALIDATE PAGE TABLE ENTRY or INVALIDATE DAT TABLE ENTRY, (2) alter bits 56-63 of a page-table entry, or (3) make a change by means of a COMPARE AND SWAP AND PURGE instruction that purges the TLB.

b. When any change is made to an attached and valid or unattached or invalid table entry other than a change to bits 56-63 of a page-table entry, each CPU which may have a TLB entry formed from that entry must be caused to purge its TLB after the change occurs and prior to the use of that entry for implicit translation by that CPU. (Note that a separate purge is unnecessary if the change was made by using INVALIDATE DAT TABLE ENTRY, INVALIDATE PAGE TABLE ENTRY, or a COMPARE AND SWAP AND PURGE instruction that purges the TLB.) In the case when the table entry is attached and valid, this rule applies when it is known that a program is not being executed that may require the entry for translation.

c. When any change is made to an invalid table entry in such a way as to allow intermediate valid values to appear in the entry, each CPU to which the entry is attached must be caused to purge its TLB after the change occurs and prior to the use of the entry for implicit address translation by that CPU.

d. When any change is made to an offset or length specified for a table, each CPU which may have a TLB entry formed from a table entry that no longer lies within its table must be caused to purge its TLB after the change occurs and prior to the use of the table for implicit translation by that CPU.

Note that when an invalid page-table entry is made valid without introducing intermediate valid values, the TLB need not be cleared in a CPU which does not have any TLB entries formed from that entry. Similarly, when an
invalid region-table or segment-table entry is made valid without introducing intermediate valid values, the TLB need not be cleared in a CPU which does not have any TLB entries formed from that validated entry and which does not have any TLB entries formed from entries in a page table attached by means of that validated entry.

The execution of PURGE TLB, COMPARE AND SWAP AND PURGE, or SET PREFIX may have an adverse effect on the performance of some models. Use of these instructions should, therefore, be minimized in conformance with the above rules.

**Address Summary**

**Addresses Translated**

Most addresses that are explicitly specified by the program and are used by the CPU to refer to storage are instruction or logical addresses and are subject to implicit translation when DAT is on. Analogously, the corresponding addresses indicated to the program on an interruption or as the result of executing an instruction are instruction or logical addresses. The operand address of LOAD REAL ADDRESS and STORE REAL ADDRESS is explicitly translated, regardless of whether the PSW specifies DAT on or off.

Translation is not applied to quantities that are formed from the values specified in the B and D fields of an instruction but that are not used to address storage. This includes operand addresses in LOAD ADDRESS, LOAD ADDRESS EXTENDED, MONITOR CALL, and the shifting instructions. This also includes the addresses in control registers 10 and 11 designating the starting and ending locations for PER.

With the exception of INSERT VIRTUAL STORAGE KEY and TEST PROTECTION, the addresses explicitly designating storage keys (operand addresses in SET STORAGE KEY EXTENDED, INSERT STORAGE KEY EXTENDED, and RESET REFERENCE BIT EXTENDED) are real addresses. Similarly, the addresses implicitly used by the CPU for such sequences as interruptions are real addresses.

The addresses used by channel programs to transfer data and to refer to CCWs or IDAWs are absolute addresses.

The handling of storage addresses associated with DIAGNOSE is model-dependent.

The processing of addresses, including dynamic address translation and prefixing, is discussed in "Address Types" on page 3-3. Prefixing, when provided, is applied after the address has been translated by means of the dynamic-address-translation facility. For a description of prefixing, see "Prefixing" on page 3-15.

**Handling of Addresses**

The handling of addresses is summarized in Figure 3-11 on page 3-56. This figure lists all addresses that are encountered by the program and specifies the address type.
### Virtual Addresses

- Address of storage operand for INSERT VIRTUAL STORAGE KEY
- Operand address in LOAD REAL ADDRESS and STORE REAL ADDRESS
- Addresses of storage operands for MOVE TO PRIMARY and MOVE TO SECONDARY
- Address stored in the doubleword at real location 168 on a program interruption for ASCE-type, region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception
- Linkage-stack-entry address in control register 15
- Backward stack-entry address in linkage-stack header entry
- Forward-section-header address in linkage-stack trailer entry
- Trap-control-block address in dispatchable-unit-control table
- Trap-save-area address and trap-program address in trap control block

### Instruction Addresses

- Instruction address in PSW
- Branch address
- Target of EXECUTE
- Address stored in the doubleword at real location 152 on a program interruption for PER
- Address placed in general register by BRANCH AND LINK, BRANCH AND SAVE, BRANCH AND SAVE AND SET MODE, BRANCH AND STACK, BRANCH IN SUBSPACE GROUP, BRANCH RELATIVE AND SAVE, BRANCH RELATIVE AND SAVE LONG, and PROGRAM CALL
- Address used in general register by BRANCH AND STACK.
- Address placed in general register by BRANCH AND SET AUTHORITY executed in reduced-authority state

### Logical Addresses

- Addresses of storage operands for instructions not otherwise specified
- Address placed in general register 1 by EDIT AND MARK and TRANSLATE AND TEST
- Addresses in general registers updated by MOVE LONG, MOVE LONG EXTENDED, COMPARE LOGICAL LONG, and COMPARE LOGICAL LONG EXTENDED
- Addresses in general registers updated by CHECKSUM, COMPARE AND FORM CODEWORD, and UPDATE TREE
- Address for TEST PENDING INTERRUPTION when the second-operand address is nonzero
- Address of parameter list of RESUME PROGRAM

---

**Figure 3-11 (Part 1 of 3). Handling of Addresses**
Real Addresses

- Address of storage key for INSERT STORAGE KEY EXTENDED, RESET REFERENCE BIT EXTENDED, and SET STORAGE KEY EXTENDED
- Address of storage operand for LOAD USING REAL ADDRESS, STORE USING REAL ADDRESS, and TEST BLOCK
- The translated address generated by LOAD REAL ADDRESS and STORE REAL ADDRESS
- Page-frame real address in page-table entry
- Trace-entry address in control register 12
- ASN-first-table origin in control register 14
- ASN-second-table origin in ASN-first-table entry
- Authority-table origin in ASN-second-table entry, except when used by access-register translation
- Linkage-table origin in primary ASN-second-table entry
- Entry-table origin in linkage-table entry
- Dispatchable-unit-control-table origin in control register 2
- Primary-ASN-second-table-entry origin in control register 5
- Base-ASN-second-table-entry origin and subspace-ASN-second-table-entry origin in dispatchable-unit control table
- ASN-second-table-entry address in entry-table entry and access-list entry

Permanently Assigned Real Addresses

- Address of the doubleword into which TEST PENDING INTERRUPTION stores when the second-operand address is zero
- Addresses of PSWs, interruption codes, and the associated information used during interruption
- Addresses used for machine-check logout and save areas
- Address of STORE FACILITY LIST operand

Addresses which Are Unpredictably Real or Absolute

- Region-first-table origin, region-second-table origin, region-third-table origin, or segment-table origin in control registers 1, 7, and 13, in access-register-specified address-space-control element, and in region-first-table entry, region-second-table entry, or region-third-table entry
- Page-table origin in segment-table entry and in INVALIDATE PAGE TABLE ENTRY
- Address of segment-table entry or page-table entry provided by LOAD REAL ADDRESS
- The dispatchable-unit or primary-space access-list origin and the authority-table origin (in the ASTE designated by the ALE used) used by access-register translation

Figure 3-11 (Part 2 of 3). Handling of Addresses
Absolute Addresses

- Prefix value
- Channel-program address in ORB
- Data address in CCW
- IDAW address in a CCW specifying indirect data addressing
- CCW address in a CCW specifying transfer in channel
- Data address in IDAW
- Measurement-block origin specified by SET CHANNEL MONITOR
- Address limit specified by SET ADDRESS LIMIT
- Addresses used by the store-status-at-address SIGNAL PROCESSOR order
- Failing-storage address stored in the doubleword at real location 248
- CCW address in SCSW

Permanently Assigned Absolute Addresses

- Addresses used for the store-status function
- Addresses of PSW and first two CCWs used for initial program loading

Addresses Not Used to Reference Storage

- PER starting address in control register 10
- PER ending address in control register 11
- Address stored in the doubleword at real location 176 for a monitor event
- Address in shift instructions and other instructions specified not to use the address to reference storage
- Real-space token origin in real-space designation

Figure 3-11 (Part 3 of 3). Handling of Addresses

Assigned Storage Locations

Figure 3-12 on page 3-65 shows the format and extent of the assigned locations in storage. The locations are used as follows.

128-131 (Real Address)

External-Interruption Parameter: During an external interruption due to service signal or the external time reference (ETR), the parameter associated with the interruption is stored at locations 128-131.

132-133 (Real Address)

CPU Address: During an external interruption due to malfunction alert, emergency signal, or external call, the CPU address associated with the source of the interruption is stored at locations 132-133. For all other external-interruption conditions, zeros are stored at locations 132-133.

134-135 (Real Address)

External-Interruption Code: During an external interruption, the interruption code is stored at locations 134-135.

136-139 (Real Address)

Supervisor-Call-Interruption Identification: During a supervisor-call interruption, the instruction-length code is stored in bit positions 5 and 6 of location 137, and the interruption code is stored at locations 138-139. Zeros are stored at location 136 and in the remaining bit positions of location 137.

140-143 (Real Address)

Program-Interruption Identification: During a program interruption, the instruction-length code is stored in bit positions 5 and 6 of location 141, and the interruption code is stored at locations 142-143. Zeros are stored at location 140 and in the remaining bit positions of location 141.
144-147 (Real Address)

**Data-Exception Code (DXC):** During a program interruption due to a data exception, the data-exception code is stored at location 147, and zeros are stored at locations 144-146. The DXC is described in "Data-Exception Code (DXC)" on page 6-14.

148-149 (Real Address)

**Monitor-Class Number:** During a program interruption due to a monitor event, the monitor-class number is stored at location 149, and zeros are stored at location 148.

150-151 (Real Address)

**PER Code:** During a program interruption due to a PER event, the PER code is stored in bit positions 0-2 and 4 of locations 150-151, and other information is or may be stored as described in "Identification of Cause" on page 4-29.

152-159 (Real Address)

**PER Address:** During a program interruption due to a PER event, the PER address is stored at locations 152-159.

160 (Real Address)

**Exception Access Identification:** During a program interruption due to an ASCE-type, region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception, an indication of the address space to which the exception applies may be stored at location 160. If the CPU was in the access-register mode and the access was an instruction fetch, including a fetch of the target of an EXECUTE instruction, zeros are stored at location 160. If the CPU was in the access-register mode but the access was an implicit reference to the linkage stack, or if the CPU was not in the access-register mode, the contents of location 160 are unpredictable.

During a program interruption due to an ALEN-translation, ALE-sequence, ASTE-validity, ASTE-sequence, or extended-authority exception recognized during access/register translation, the number of the access register used is stored in bit positions 4-7 of location 160, and zeros are stored in bit positions 0-3. During a program interruption due to an ASTE-validity or ASTE-sequence exception recognized during a subspace-replacement operation, all zeros are stored at location 160.

During a program interruption due to an ASTE-instance exception recognized due to use of the ASN-and-LX-reuse facility, (1) a one is stored in bit position 2, and zeros are stored in bit positions 0, 1, and 3-7, if the exception was recognized after primary ASN translation in PROGRAM TRANSFER WITH INSTANCE or PROGRAM RETURN, or (2) a one is stored in bit position 3, and zeros are stored in bit positions 0-2 and 4-7, if the exception was recognized after secondary ASN translation in SET SECONDARY ASN WITH INSTANCE or PROGRAM RETURN.

During a program interruption due to a protection exception, information is stored at location 160 as described in "Suppression on Protection" on page 3-12.

161 (Real Address)

**PER Access Identification:** During a program interruption due to a PER storage-alteration event, an indication of the address space to which the event applies may be stored at location 161. If the access used an AR-specified address-space-control element, the number of the access register used is stored in bit positions 4-7 of location 161, and zeros are stored in bit positions 0-3. The contents of location 161 are unpredictable if (1) the CPU was in the
access-register mode but the access was an implicit reference to the linkage stack or (2) the CPU was not in the access-register mode.

162 (Real Address)

Operand Access Identification: During a program interruption due to a page-translation exception recognized by the MOVE PAGE instruction, the contents of the R₁ field of the instruction are stored in bit positions 0-3 of location 162, and the contents of the R₂ field are stored in bit positions 4-7. If the page-translation exception was recognized during the execution of an instruction other than MOVE PAGE, or if an ASCE-type, region-first-translation, region-second-translation, region-third-translation, or segment-translation exception was recognized, the contents of location 162 are unpredictable.

163 (Absolute Address)

Store-Status Architectural-Mode Identification: During the execution of the store-status operation, zeros are stored in bit positions 0-6 of location 163, and a one is stored in bit position 7. A zero stored in bit position 7 indicates the ESA/390 architectural mode, and a one indicates the z/Architecture architectural mode.

163 (Real Address)

Machine-Check Architectural-Mode Identification: During a machine-check interruption, zeros are stored in bit positions 0-6 of location 163, and a one is stored in bit position 7. A zero stored in bit position 7 indicates the ESA/390 architectural mode, and a one indicates the z/Architecture architectural mode.

168-175 (Real Address)

Translation-Exception Identification: During a program interruption due to an ASCE-type, region-first-translation, region-second-translation, region-third-translation, segment-translation, or page-translation exception, bits 0-51 of the virtual address causing the exception are stored in bit positions 0-51 of locations 168-175. This address is sometimes referred to as the translation-exception address. Bits 52-60 of locations 168-175 are unpredictable. If the exception was a page-translation exception that was recognized during the execution of MOVE PAGE, bit 61 of locations 168-175 is set to one. If the exception was a page-translation exception recognized during the execution of an instruction other than MOVE PAGE, bit 61 is set to zero. If the exception was an ASCE-type, region-first-translation, region-second-translation, region-third-translation, or segment-translation exception, bit 61 of locations 168-175 is unpredictable. See the definition of real location 162 for related information.

Bits 62 and 63 of locations 168-175 are set to identify the address-space-control element (ASCE) used in the translation, as follows:

<table>
<thead>
<tr>
<th>Bit 62</th>
<th>Bit 63</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Primary ASCE was used.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>CPU was in the access-register mode, and either the access was an instruction fetch or it was a storage-operand reference that used an AR-specified ASCE (the access was not an implicit reference to the linkage stack). The exception access id, real location 160, can be examined to determine the ASCE used. However, if the primary, secondary, or home ASCE was used, bits 62 and 63 may be set to 00, 10, or 11, respectively, instead of to 01.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Secondary ASCE was used.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Home ASCE was used (includes the case of an implicit reference to the linkage stack).</td>
</tr>
</tbody>
</table>

The CPU may avoid setting bits 62 and 63 to 01 by recognizing that the access was an instruction fetch, that access-list-entry token 00000000 or 00000001 hex was used, or that the access-list-entry token designated, through an
access-list entry, an ASN-second-table entry containing an ASCE equal to the primary ASCE, secondary ASCE, or home ASCE.

During a program interruption due to an AFX-translation, ASX-translation, primary-authority, or secondary-authority exception, the ASN being translated is stored at locations 174 and 175, zeros are stored at locations 172 and 173, and the contents of locations 168-171 remain unchanged.

During a program interruption due to a space-switch event, an identification of the old instruction space is stored at locations 174 and 175, the old instruction-space space-switch-event-control bit is placed in bit position 0 and zeros are placed in bit positions 1-15 of locations 172 and 173, and the contents of locations 168-171 remain unchanged. The identification and bit stored are as follows:

- If the CPU was in the primary-space, secondary-space, or access-register mode before the operation, the old PASN, bits 48-63 of control register 4 before the operation, is stored at locations 174 and 175, and the old primary space-switch-event-control bit, bit 57 of control register 1 before the operation, is placed in bit position 0 of locations 172 and 173.

- If the CPU was in the home-space mode before the operation, zeros are stored at locations 174 and 175, and the home space-switch-event-control bit, bit 57 of control register 13, is placed in bit position 0 of locations 172 and 173.

During a program interruption due to an EX-translation or LX-translation exception recognized by PROGRAM CALL when ASN-and-LX reuse is not installed or is not enabled by a one value of the ASN-and-LX-reuse control in control register 0, bits 44-63 of the second-operand address used by PROGRAM CALL (a 20-bit PC number), with 12 zeros appended on the left, are stored at locations 172-175, and the contents of locations 168-171 remain unchanged.

During a program interruption due to an EX-translation, LFX-translation, LSTE-sequence, or LSX-translation exception recognized by PROGRAM CALL when ASN-and-LX reuse is installed and enabled and bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the second-operand address (a 20-bit PC number), with 12 zeros appended on the left, are stored at locations 172-175. If bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored at locations 172-175. In either of these cases, the contents of locations 168-171 remain unchanged.

During a program interruption due to a protection exception, information is stored at locations 168-175 as described in "Suppression on Protection" on page 3-12.

176-183 (Real Address)

Monitor Code: During a program interruption due to a monitor event, the monitor code is stored at locations 176-183.

184-187 (Real Address)

Subsystem-Identification Word: During an I/O interruption, the subsystem-identification word is stored at locations 184-187.

188-191 (Real Address)

I/O-Interruption Parameter: During an I/O interruption, the interruption parameter from the associated subchannel is stored at locations 188-191.

192-195 (Real Address)

I/O-Interruption-Identification Word: During an I/O interruption, the I/O-interruption-identification word, which further identifies the source of the I/O interruption, is stored at locations 192-195.
200-203 (Real Address)

**STFL Facility List**: The STORE FACILITY LIST instruction stores information at real locations 200-203. The information describes which facilities are provided by the CPU. See the definition of STORE FACILITY LIST in Chapter 10, “Control Instructions,” for a description of the information stored.

232-239 (Real Address)

**Machine-Check-Interruption Code**: During a machine-check interruption, the machine-check-interruption code is stored at locations 232-239.

244-247 (Real Address)

**External-Damage Code**: During a machine-check interruption due to certain external-damage conditions, depending on the model, an external-damage code may be stored at locations 244-247.

248-255 (Real Address)

**Failing-Storage Address**: During a machine-check interruption, a 64-bit failing-storage address may be stored at locations 248-255.

288-303 (Real Address)

**Restart Old PSW**: The current PSW is stored as the old PSW at locations 288-303 during a restart interruption.

304-319 (Real Address)

**External Old PSW**: The current PSW is stored as the old PSW at locations 304-319 during an external interruption.

320-335 (Real Address)

**Supervisor-Call Old PSW**: The current PSW is stored as the old PSW at locations 320-335 during a supervisor-call interruption.

336-351 (Real Address)

**Program Old PSW**: The current PSW is stored as the old PSW at locations 336-351 during a program interruption.

352-367 (Real Address)

**Machine-Check Old PSW**: The current PSW is stored as the old PSW at locations 352-367 during a machine-check interruption.

368-383 (Real Address)

**Input/Output Old PSW**: The current PSW is stored as the old PSW at locations 368-383 during an I/O interruption.

416-431 (Real Address)

**Restart New PSW**: The new PSW is fetched from locations 416-431 during a restart interruption.

432-447 (Real Address)

**External New PSW**: The new PSW is fetched from locations 432-447 during an external interruption.

448-463 (Real Address)

**Supervisor-Call New PSW**: The new PSW is fetched from locations 448-463 during a supervisor-call interruption.

464-479 (Real Address)

**Program New PSW**: The new PSW is fetched from locations 464-479 during a program interruption.

480-495 (Real Address)

**Machine-Check New PSW**: The new PSW is fetched from locations 480-495 during a machine-check interruption.

496-511 (Real Address)

**Input/Output New PSW**: The new PSW is fetched from locations 496-511 during an I/O interruption.

4544-4607 (Real Address)

**Available for Programming**: Locations 4544-4607 are available for use by programming.

4608-4735 (Absolute Address)

**Store-Status Floating-Point-Register Save Area**: During the execution of the store-status operation, the contents of the floating-point registers are stored at locations 4608-4735.

4608-4735 (Real Address)

**Machine-Check Floating-Point-Register Save Area**: During a machine-check interruption, the contents of the floating-
point registers are stored at locations 4608-4735.

4736-4863 (Absolute Address)

**Store-Status General-Register Save Area:** During the execution of the store-status operation, the contents of the general registers are stored at locations 4736-4863.

4736-4863 (Real Address)

**Machine-Check General-Register Save Area:** During a machine-check interruption, the contents of the general registers are stored at locations 4736-4863.

4864-4879 (Absolute Address)

**Store-Status PSW Save Area:** During the execution of the store-status operation, the contents of the current PSW are stored at locations 4864-4879.

4864-4879 (Real Address)

**Fixed-Logout Area:** Depending on the model, logout information may be stored at locations 4864-4879 during a machine-check interruption.

4888-4891 (Absolute Address)

**Store-Status Prefix Save Area:** During the execution of the store-status operation, the contents of the prefix register are stored at locations 4888-4891.

4892-4895 (Absolute Address)

**Store-Status Floating-Point-Control-Register Save Area:** During the execution of the store-status operation, the contents of the floating-point control register are stored at locations 4892-4895.

4892-4895 (Real Address)

**Machine-Check Floating-Point-Control-Register Save Area:** During a machine-check interruption, the contents of the floating-point control register are stored at locations 4892-4895.

4900-4903 (Absolute Address)

**Store-Status TOD-Programmable-Register Save Area:** During the execution of the store-status operation, the contents of the TOD programmable register are stored at locations 4900-4903.

4900-4903 (Real Address)

**Machine-Check TOD-Programmable-Register Save Area:** During a machine-check interruption, the contents of the TOD programmable register are stored at locations 4904-4911.

4904-4911 (Absolute Address)

**Store-Status CPU-Timer Save Area:** During the execution of the store-status operation, the contents of the CPU timer are stored at locations 4904-4911.

4904-4911 (Real Address)

**Machine-Check CPU-Timer Save Area:** During a machine-check interruption, the contents of the CPU timer are stored at locations 4904-4911.

4913-4919 (Absolute Address)

**Store-Status Clock-Comparator Save Area:** During the execution of the store-status operation, the contents of bit positions 0-55 of the clock comparator are stored at locations 4913-4919. When this store occurs, zeros are stored at location 4912.

4913-4919 (Real Address)

**Machine-Check Clock-Comparator Save Area:** During a machine-check interruption, the contents of bit positions 0-55 of the clock comparator are stored at locations 4913-4919. When this store occurs, zeros are stored at location 4912.

4928-4991 (Absolute Address)

**Store-Status Access-Register Save Area:** During the execution of the store-status operation, the contents of the access registers are stored at locations 4928-4991.

4928-4991 (Real Address)

**Machine-Check Access-Register Save Area:** During a machine-check interruption, the contents of the access registers are stored at locations 4928-4991.
4992-5119 (Absolute Address)

*Store-Status Control-Register Save Area:* During the execution of the store-status operation, the contents of the control registers are stored at locations 4992-5119.

4992-5119 (Real Address)

*Machine-Check Control-Register Save Area:* During a machine-check interruption, the contents of the control registers are stored at locations 4992-5119.

**Programming Notes:**

1. When the CPU is in the access-register mode, some instructions, such as MVCL, which address operands in more than one address space, may cause a storage-alteration PER event in one address space concurrently with a region-translation, segment-translation, or page-translation exception in another address space. The access registers used to cause these conditions in such a case are different. In order to identify both access registers, two access identifications, namely the exception access identification and the PER access identification, are provided.

2. The store-status and machine-check architectural-mode identifications at absolute and real locations 163, respectively, indicate that the CPU is in the z/Architecture architectural mode. When z/Architecture is installed on the CPU but the CPU is in the ESA/390 mode, the store-status and machine-check-interruption operations store zero at location 163.
<table>
<thead>
<tr>
<th>Hex Dec</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>4 4</td>
<td></td>
</tr>
<tr>
<td>8 8</td>
<td></td>
</tr>
<tr>
<td>C 12</td>
<td></td>
</tr>
<tr>
<td>10 16</td>
<td></td>
</tr>
<tr>
<td>14 20</td>
<td></td>
</tr>
<tr>
<td>18 24</td>
<td></td>
</tr>
<tr>
<td>1C 28</td>
<td></td>
</tr>
<tr>
<td>20 32</td>
<td></td>
</tr>
<tr>
<td>24 36</td>
<td></td>
</tr>
<tr>
<td>28 40</td>
<td></td>
</tr>
<tr>
<td>2C 44</td>
<td></td>
</tr>
<tr>
<td>30 48</td>
<td></td>
</tr>
<tr>
<td>34 52</td>
<td></td>
</tr>
<tr>
<td>38 56</td>
<td></td>
</tr>
<tr>
<td>3C 60</td>
<td></td>
</tr>
<tr>
<td>40 64</td>
<td></td>
</tr>
<tr>
<td>44 68</td>
<td></td>
</tr>
<tr>
<td>48 72</td>
<td></td>
</tr>
<tr>
<td>4C 76</td>
<td></td>
</tr>
<tr>
<td>50 80</td>
<td></td>
</tr>
<tr>
<td>54 84</td>
<td></td>
</tr>
<tr>
<td>58 88</td>
<td></td>
</tr>
<tr>
<td>5C 92</td>
<td></td>
</tr>
<tr>
<td>60 96</td>
<td></td>
</tr>
<tr>
<td>64 100</td>
<td></td>
</tr>
<tr>
<td>68 104</td>
<td></td>
</tr>
<tr>
<td>6C 108</td>
<td></td>
</tr>
<tr>
<td>70 112</td>
<td></td>
</tr>
<tr>
<td>74 116</td>
<td></td>
</tr>
<tr>
<td>78 120</td>
<td></td>
</tr>
<tr>
<td>7C 124</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 3-12 (Part 1 of 6). Assigned Storage Locations*
### Hex Dec | Fields
--- | ---
80 128 | External-Interruption Parameter
84 132 | CPU Address
88 136 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ILC 0 | SVC-Interruption Code
8C 140 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | ILC 0 | Program-Interruption Code
90 144 | Data-Exception Code
94 148 | Monitor-Class Number
98 152 | PER Address
A0 160 | Exc. Access ID
A4 164 | Translation-Exception Identification
48 168 | Monitor Code
B0 176 | Subsystem-Identification Word
B4 180 | I/O-Interruption Parameter
B8 184 | I/O-Interruption-Identification Word
C4 196 | STFL Facility List
CC 204 | Machine-Check Interruption Code
D0 208 | External-Damage Code
D4 212 | Failing-Storage Address
D8 216 | Failing-Storage Address
DC 220 | Failing-Storage Address
E0 224 | Failing-Storage Address
E4 228 | Failing-Storage Address
F8 248 | Failing-Storage Address
FC 252 | Failing-Storage Address

*Figure 3-12 (Part 2 of 6). Assigned Storage Locations*
<table>
<thead>
<tr>
<th>Hex Dec</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>256</td>
</tr>
<tr>
<td>104</td>
<td>260</td>
</tr>
<tr>
<td>108</td>
<td>264</td>
</tr>
<tr>
<td>10C</td>
<td>268</td>
</tr>
<tr>
<td>110</td>
<td>272</td>
</tr>
<tr>
<td>114</td>
<td>276</td>
</tr>
<tr>
<td>118</td>
<td>280</td>
</tr>
<tr>
<td>11C</td>
<td>284</td>
</tr>
<tr>
<td>120</td>
<td>Restart Old PSW</td>
</tr>
<tr>
<td>124</td>
<td>292</td>
</tr>
<tr>
<td>128</td>
<td>296</td>
</tr>
<tr>
<td>12C</td>
<td>300</td>
</tr>
<tr>
<td>130</td>
<td>External Old PSW</td>
</tr>
<tr>
<td>134</td>
<td>308</td>
</tr>
<tr>
<td>138</td>
<td>312</td>
</tr>
<tr>
<td>13C</td>
<td>316</td>
</tr>
<tr>
<td>140</td>
<td>Supervisor-Call Old PSW</td>
</tr>
<tr>
<td>144</td>
<td>324</td>
</tr>
<tr>
<td>148</td>
<td>328</td>
</tr>
<tr>
<td>14C</td>
<td>332</td>
</tr>
<tr>
<td>150</td>
<td>Program Old PSW</td>
</tr>
<tr>
<td>154</td>
<td>340</td>
</tr>
<tr>
<td>158</td>
<td>344</td>
</tr>
<tr>
<td>15C</td>
<td>348</td>
</tr>
<tr>
<td>160</td>
<td>Machine-Check Old PSW</td>
</tr>
<tr>
<td>164</td>
<td>352</td>
</tr>
<tr>
<td>168</td>
<td>360</td>
</tr>
<tr>
<td>16C</td>
<td>364</td>
</tr>
<tr>
<td>170</td>
<td>Input/Output Old PSW</td>
</tr>
<tr>
<td>174</td>
<td>372</td>
</tr>
<tr>
<td>178</td>
<td>376</td>
</tr>
<tr>
<td>17C</td>
<td>380</td>
</tr>
</tbody>
</table>

Figure 3-12 (Part 3 of 6). Assigned Storage Locations
<table>
<thead>
<tr>
<th>Hex Dec</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>180 384</td>
<td></td>
</tr>
<tr>
<td>184 388</td>
<td></td>
</tr>
<tr>
<td>188 392</td>
<td></td>
</tr>
<tr>
<td>18C 396</td>
<td></td>
</tr>
<tr>
<td>190 400</td>
<td></td>
</tr>
<tr>
<td>194 404</td>
<td></td>
</tr>
<tr>
<td>198 408</td>
<td></td>
</tr>
<tr>
<td>19C 412</td>
<td></td>
</tr>
<tr>
<td>1A0 416</td>
<td>Restart New PSW</td>
</tr>
<tr>
<td>1A4 420</td>
<td></td>
</tr>
<tr>
<td>1A8 424</td>
<td></td>
</tr>
<tr>
<td>1AC 428</td>
<td></td>
</tr>
<tr>
<td>1B0 432</td>
<td>External New PSW</td>
</tr>
<tr>
<td>1B4 436</td>
<td></td>
</tr>
<tr>
<td>1B8 440</td>
<td></td>
</tr>
<tr>
<td>1BC 444</td>
<td></td>
</tr>
<tr>
<td>1C0 448</td>
<td>Supervisor-Call New PSW</td>
</tr>
<tr>
<td>1C4 452</td>
<td></td>
</tr>
<tr>
<td>1CB 456</td>
<td></td>
</tr>
<tr>
<td>1CC 460</td>
<td></td>
</tr>
<tr>
<td>1D0 464</td>
<td>Program New PSW</td>
</tr>
<tr>
<td>1D4 468</td>
<td></td>
</tr>
<tr>
<td>1DB 472</td>
<td></td>
</tr>
<tr>
<td>1DC 476</td>
<td></td>
</tr>
<tr>
<td>1E0 480</td>
<td>Machine-Check New PSW</td>
</tr>
<tr>
<td>1E4 484</td>
<td></td>
</tr>
<tr>
<td>1EB 488</td>
<td></td>
</tr>
<tr>
<td>1EC 492</td>
<td></td>
</tr>
<tr>
<td>1F0 496</td>
<td>Input/Output New PSW</td>
</tr>
<tr>
<td>1F4 500</td>
<td></td>
</tr>
<tr>
<td>1FB 504</td>
<td></td>
</tr>
<tr>
<td>1FC 508</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 3-12 (Part 4 of 6). Assigned Storage Locations*
<table>
<thead>
<tr>
<th>Hex Dec</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 4096</td>
<td></td>
</tr>
<tr>
<td>1004 4100</td>
<td></td>
</tr>
<tr>
<td>1008 4104</td>
<td></td>
</tr>
<tr>
<td>100C 4108</td>
<td></td>
</tr>
<tr>
<td>1010 4112</td>
<td></td>
</tr>
<tr>
<td>1014 4116</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(448 bytes)</td>
</tr>
<tr>
<td>11A8 4520</td>
<td></td>
</tr>
<tr>
<td>11AC 4524</td>
<td></td>
</tr>
<tr>
<td>11B0 4528</td>
<td></td>
</tr>
<tr>
<td>11B4 4532</td>
<td></td>
</tr>
<tr>
<td>11B8 4536</td>
<td></td>
</tr>
<tr>
<td>11BC 4540</td>
<td></td>
</tr>
<tr>
<td>11C0 4544</td>
<td>Available for Use by Programming</td>
</tr>
<tr>
<td>11C4 4548</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(64 bytes)</td>
</tr>
<tr>
<td>11F8 4600</td>
<td></td>
</tr>
<tr>
<td>11FC 4604</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-12 (Part 5 of 6). Assigned Storage Locations
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200</td>
<td>4608</td>
<td>Store-Status Floating-Point-Register Save Area; or Machine-Check Floating-Point-Register Save Area</td>
</tr>
<tr>
<td>1204</td>
<td>4612</td>
<td>(128 bytes)</td>
</tr>
<tr>
<td>1278</td>
<td>4728</td>
<td></td>
</tr>
<tr>
<td>127C</td>
<td>4732</td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td>4736</td>
<td>Store-Status General-Register Save Area; or Machine-Check General-Register Save Area</td>
</tr>
<tr>
<td>1284</td>
<td>4740</td>
<td>(128 bytes)</td>
</tr>
<tr>
<td>12FB</td>
<td>4856</td>
<td></td>
</tr>
<tr>
<td>12FC</td>
<td>4860</td>
<td></td>
</tr>
<tr>
<td>1300</td>
<td>4864</td>
<td>Store-Status PSW Save Area; or Fixed-Logout Area</td>
</tr>
<tr>
<td>1304</td>
<td>4868</td>
<td></td>
</tr>
<tr>
<td>1308</td>
<td>4872</td>
<td></td>
</tr>
<tr>
<td>130C</td>
<td>4876</td>
<td></td>
</tr>
<tr>
<td>1310</td>
<td>4880</td>
<td></td>
</tr>
<tr>
<td>1314</td>
<td>4884</td>
<td></td>
</tr>
<tr>
<td>1318</td>
<td>4888</td>
<td>Store-Status Prefix Save Area</td>
</tr>
<tr>
<td>131C</td>
<td>4892</td>
<td>Store-Status FP-Ctl-Reg Save Area; or MC FP-Ctl-Reg Save Area</td>
</tr>
<tr>
<td>1320</td>
<td>4896</td>
<td></td>
</tr>
<tr>
<td>1324</td>
<td>4900</td>
<td>Store-Status TOD Prog Reg Save Area; or MC TOD Prog Reg S A</td>
</tr>
<tr>
<td>1328</td>
<td>4904</td>
<td>Store-Status CPU-Timer Save Area; or Machine-Check CPU-Timer Save Area</td>
</tr>
<tr>
<td>132C</td>
<td>4908</td>
<td></td>
</tr>
<tr>
<td>1330</td>
<td>4912</td>
<td>Store-Status Clock-Comparator Bits 0-55 Save Area; or Machine-Check Clock-Comparator Bits 0-55 Save Area</td>
</tr>
<tr>
<td>1334</td>
<td>4916</td>
<td></td>
</tr>
<tr>
<td>1338</td>
<td>4920</td>
<td></td>
</tr>
<tr>
<td>133C</td>
<td>4924</td>
<td></td>
</tr>
<tr>
<td>1340</td>
<td>4928</td>
<td>Store-Status Access-Register Save Area; or Machine-Check Access-Register Save Area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64 bytes)</td>
</tr>
<tr>
<td>137C</td>
<td>4988</td>
<td></td>
</tr>
<tr>
<td>1380</td>
<td>4992</td>
<td>Store-Status Control-Register Save Area; or Machine-Check Control-Register Save Area</td>
</tr>
<tr>
<td>1384</td>
<td>4996</td>
<td>(128 bytes)</td>
</tr>
<tr>
<td>13FB</td>
<td>5112</td>
<td></td>
</tr>
<tr>
<td>13FC</td>
<td>5116</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-12 (Part 6 of 6). Assigned Storage Locations
# Chapter 4. Control

Stopped, Operating, Load, and Check-Stop States .................................................. 4-1
Stopped State ............................................................................................................ 4-2
Operating State ....................................................................................................... 4-2
Load State ................................................................................................................ 4-2
Check-Stop State ..................................................................................................... 4-3
Program-Status Word ............................................................................................... 4-3
Program-Status-Word Format .................................................................................. 4-5
Control Registers ...................................................................................................... 4-7
Tracing ......................................................................................................................... 4-10
Control-Register Allocation ...................................................................................... 4-14
Trace Entries ............................................................................................................. 4-14
Operation ..................................................................................................................... 4-27
Program-Event Recording .......................................................................................... 4-27
Control-Register Allocation and Address-Space-Control Element ....................... 4-28
Operation ..................................................................................................................... 4-29
Identification of Cause ............................................................................................... 4-29
Priority of Indication ................................................................................................. 4-32
Storage-Area Designation ......................................................................................... 4-33
PER Events .................................................................................................................. 4-34
Successful Branching ................................................................................................. 4-34
Instruction Fetching .................................................................................................... 4-34
Storage Alteration ....................................................................................................... 4-35
Store Using Real Address .......................................................................................... 4-35
Indication of PER Events Concurrently with Other Interruption Conditions ....... 4-35
Timing .......................................................................................................................... 4-40

This chapter describes in detail the facilities for controlling, measuring, and recording the operation of one or more CPUs.

## Stopped, Operating, Load, and Check-Stop States

The stopped, operating, load, and check-stop states are four mutually exclusive states of the CPU. When the CPU is in the stopped state, instructions and interruptions, other than the restart interruption, are not executed. In the operating state, the CPU executes instructions and takes interruptions, subject to the control of the program-status word (PSW) and control registers, and in the manner specified by the setting of the operator-facility rate control. The CPU is in the load state during the initial-program-loading operation of ESA/390. The CPU enters the check-stop state only as the result of machine malfunctions.

A change between these four CPU states can be effected by use of the operator facilities or by acceptance of certain SIGNAL PROCESSOR orders addressed to that CPU. The states are not controlled or identified by bits in the PSW. The stopped, load, and check-stop states are indicated to the operator by means of the manual indicator, load indicator, and check-stop indicator, respectively. These three indicators are off when the CPU is in the operating state.

The CPU timer is updated when the CPU is in the operating state or the load state. The TOD clock is not affected by the state of any CPU.
Stopped State

The CPU changes from the operating state to the stopped state by means of the stop function. The stop function is performed when:

- The stop key is activated while the CPU is in the operating state.
- The CPU accepts a stop or stop-and-store-status order specified by a SIGNAL PROCESSOR instruction addressed to this CPU while it is in the operating state.
- The CPU has finished the execution of a unit of operation initiated by performing the start function with the rate control set to the instruction-step position.

When the stop function is performed, the transition from the operating to the stopped state occurs at the end of the current unit of operation. When the wait-state bit of the PSW is one, the transition takes place immediately, provided no interruptions are pending for which the CPU is enabled. In the case of interruptible instructions, the amount of data processed in a unit of operation depends on the particular instruction and may depend on the model.

Before entering the stopped state by means of the stop function, all pending allowed interruptions occur while the CPU is still in the operating state. They cause the old PSW to be stored and the new PSW to be fetched before the stopped state is entered. While the CPU is in the stopped state, interruption conditions remain pending.

The CPU is also placed in the stopped state when:

- CPU reset is completed. However, when the reset operation is performed as part of initial program loading for this CPU, then the CPU is placed in the load state and does not necessarily enter the stopped state.
- An address comparison indicates equality and stopping on the match is specified.

The execution of resets is described in "Resets" on page 4-47 and address comparison is described in "Address-Compare Controls" on page 12-1.

If the CPU is in the stopped state when an INVAL-IDATE PAGE TABLE ENTRY instruction is executed on another CPU in the configuration, the clearing of TLB entries is completed before the CPU leaves the stopped state.

Operating State

The CPU changes from the stopped state to the operating state by means of the start function or when a restart interruption (see Chapter 6, "Interruptions") occurs.

The start function is performed if the CPU is in the stopped state and (1) the start key associated with that CPU is activated or (2) that CPU accepts the start order specified by a SIGNAL PROCESSOR instruction addressed to that CPU. The effect of performing the start function is unpredictable when the stopped state has been entered by means of a reset.

When the rate control is set to the process position and the start function is performed, the CPU starts operating at normal speed. When the rate control is set to the instruction-step position and the wait-state bit is zero, one instruction or, for interruptible instructions, one unit of operation is executed, and all pending allowed interruptions occur before the CPU returns to the stopped state. When the rate control is set to the instruction-step position and the wait-state bit is one, the start function does not cause an instruction to be executed, but all pending allowed interruptions occur before the CPU returns to the stopped state.

Load State

The CPU enters the load state when the load-normal or load-clear key is activated. (See "Initial Program Loading" on page 4-54. See also "Initial Program Loading" on page 17-17.) This sets the architectural mode to the ESA/390 mode. For ease of reference, the additional elements of the description of the ESA/390 load state are given below.

If a CCW-type initial-program-loading operation is completed successfully, the CPU changes from the load state to the operating state, provided the rate control is set to the process position; if the rate control is set to the instruction-step position, the CPU changes from the load state to the stopped state.
Check-Stop State

The check-stop state, which the CPU enters on certain types of machine malfunction, is described in [Chapter 11, “Machine-Check Handling.”] The CPU leaves the check-stop state when CPU reset is performed.

Programming Notes:

1. Except for the relationship between execution time and real time, the execution of a program is not affected by stopping the CPU.

2. When, because of a machine malfunction, the CPU is unable to end the execution of an instruction, the stop function is ineffective, and a reset function has to be invoked instead. A similar situation occurs when an unending string of interruptions results from a PSW with a PSW-format error of the type that is recognized early, or from a persistent interruption condition, such as one due to the CPU timer.

3. Pending I/O operations may be initiated, and active I/O operations continue to suspension or completion, after the CPU enters the stopped state. The interruption conditions due to suspension or completion of I/O operations remain pending when the CPU is in the stopped state.

Program-Status Word

The current program-status word (PSW) in the CPU contains information required for the execution of the currently active program. The PSW is 128 bits in length and includes the instruction address, condition code, and other control fields. In general, the PSW is used to control instruction sequencing and to hold and indicate much of the status of the CPU in relation to the program currently being executed. Additional control and status information is contained in control registers and permanently assigned storage locations.

The status of the CPU can be changed by loading a new PSW or part of a PSW.

Control is switched during an interruption of the CPU by storing the current PSW, so as to preserve the status of the CPU, and then loading a new PSW.

Execution of LOAD PSW or LOAD PSW EXTENDED, or the successful conclusion of the initial-program-loading sequence, introduces a new PSW. The instruction address is updated by sequential instruction execution and replaced by successful branches. Other instructions are provided which operate on a portion of the PSW. [Figure 4-1 on page 4-4] summarizes these instructions.

A new or modified PSW becomes active (that is, the information introduced into the current PSW assumes control over the CPU) when the interruption or the execution of an instruction that changes the PSW is completed. The interruption for PER associated with an instruction that changes the PSW occurs under control of the PER mask that is effective at the beginning of the operation.

Bits 0-7 of the PSW are collectively referred to as the system mask.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>System Mask (PSW Bits 0-7)</th>
<th>PSW Key (PSW Bits 8-11)</th>
<th>Problem State (PSW Bit 15)</th>
<th>Address-Space Control (PSW Bits 16-17)</th>
<th>Condition Code and Program Mask (PSW Bits 18-23)</th>
<th>Basic Addressing Mode (PSW Bit 32)</th>
<th>Extended Addressing Mode (PSW Bit 31)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH AND LINK</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>24AM</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH AND SAVE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31AM</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH AND SAVE AND SET MODE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH AND SET AUTHORITY</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH AND SET MODE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH AND STACK</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH IN SUBSPACE GROUP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH RELATIVE AND SAVE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BRANCH RELATIVE AND SAVE LONG</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>EXTRACT PSW</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>INSERT PROGRAM MASK</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INSERT PSW KEY</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INSERT ADDRESS SPACE CONTROL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Basic PROGRAM CALL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stacking PROGRAM CALL</td>
<td>Yes</td>
<td>-</td>
<td>PKC</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>PROGRAM RETURN</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PROGRAM TRANSFER</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RESUME PROGRAM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SET ADDRESS SPACE CONTROL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SET ADDRESSING MODE</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SET PROGRAM MASK</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SET PSW KEY FROM ADDRESS</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SET SYSTEM MASK</td>
<td>-</td>
<td>Yes</td>
<td>ANDs</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>STORE THEN AND SYSTEM MASK</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>STORE THEN OR SYSTEM MASK</td>
<td>Yes</td>
<td>ORs</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TRAP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation:**

1. The action takes place only if the associated R field in the instruction is nonzero.

2. In the reduced-authority state, the action takes place only if the R1 field in the instruction is nonzero.

3. The action also takes place in the 64-bit addressing mode if the R1 field in the instruction is zero.

4. PROGRAM RETURN does not change the PER mask.

5. PROGRAM TRANSFER does not change the problem-state bit from one to zero.

Figure 4-1 (Part 1 of 2). Operations on PSW Fields
BAM The basic-addressing-mode bit is saved or set in the 24-bit or 31-bit addressing mode.

ANDs The logical AND of the immediate field in the instruction and the current system mask replaces the current system mask.

ORs The logical OR of the immediate field in the instruction and the current system mask replaces the current system mask.

PKC When the PSW-key-control bit, bit 131 of the entry-table entry, is zero, the PSW key remains unchanged. When the PSW-key-control bit is one, the PSW key is set with the entry key, bits 136-139 of the entry-table entry.

24AM The condition code and program mask are saved in the 24-bit addressing mode.

31AM The basic-addressing-mode bit is saved in the 31-bit addressing mode.

Figure 4-1 (Part 2 of 2). Operations on PSW Fields

Programming Note: A summary of the operations which save or set the problem state, addressing mode, and instruction address is contained in “Subroutine Linkage without the Linkage Stack” on page 5-11.

Program-Status-Word Format

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| R | O | O | O | O | T | E | X | Key | O | M | W | P | A | S | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C |
| 0 | 5 | 8 | 12 | 16 | 18 | 20 | 24 | 31 |

<table>
<thead>
<tr>
<th>32</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Instruction Address

96-127

Instructions

Chapter 4. Control
key specified as an operand is used instead of the PSW key.

**Machine-Check Mask (M):** Bit 13 controls whether the CPU is enabled for interruption by machine-check conditions. When the bit is zero, a machine-check interruption cannot occur. When the bit is one, machine-check interruptions due to system damage and instruction-processing damage are permitted, but interruptions due to other machine-check-subclass conditions are subject to the subclass-mask bits in control register 14.

**Wait State (W):** When bit 14 is one, the CPU is waiting; that is, no instructions are processed by the CPU, but interruptions may take place. When bit 14 is zero, instruction fetching and execution occur in the normal manner. The wait indicator is on when the bit is one.

**Problem State (P):** When bit 15 is one, the CPU is in the problem state. When bit 15 is zero, the CPU is in the supervisor state. In the supervisor state, all instructions are valid. In the problem state, only those instructions are valid that provide meaningful information to the problem program and that cannot affect system integrity; such instructions are called unprivileged instructions. The instructions that are never valid in the problem state are called privileged instructions. When a CPU in the problem state attempts to execute a privileged instruction, a privileged-operation exception is recognized. Another group of instructions, called semiprivileged instructions, are executed by a CPU in the problem state only if specific authority tests are met; otherwise, a privileged-operation exception or a special-operation exception is recognized.

**Address-Space Control (AS):** Bits 16 and 17, in conjunction with PSW bit 5, control the translation mode. See “Translation Modes” on page 3-35.

**Condition Code (CC):** Bits 18 and 19 are the two bits of the condition code. The condition code is set to 0, 1, 2, or 3, depending on the result obtained in executing certain instructions. Most arithmetic and logical operations, as well as some other operations, set the condition code. The instruction BRANCH ON CONDITION can specify any selection of the condition-code values as a criterion for branching. A table in Appendix C summarizes the condition-code values that may be set for all instructions which set the condition code of the PSW.

**Program Mask:** Bits 20-23 are the four program-mask bits. Each bit is associated with a program exception, as follows:

<table>
<thead>
<tr>
<th>Program-Mask Bit</th>
<th>Program Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Fixed-point overflow</td>
</tr>
<tr>
<td>21</td>
<td>Decimal overflow</td>
</tr>
<tr>
<td>22</td>
<td>HFP exponent underflow</td>
</tr>
<tr>
<td>23</td>
<td>HFP significance</td>
</tr>
</tbody>
</table>

When the mask bit is one, the exception results in an interruption. When the mask bit is zero, no interruption occurs. The setting of the HFP-exponent-underflow-mask bit or the HFP-significance-mask bit also determines the manner in which the operation is completed when the corresponding exception occurs.

**Extended Addressing Mode (EA):** Bit 31 controls the size of effective addresses and effective-address generation in conjunction with bit 32, the basic-addressing-mode bit. When bit 31 is zero, the addressing mode is controlled by bit 32. When bits 31 and 32 are both one, 64-bit addressing is specified.

**Basic Addressing Mode (BA):** Bits 31 and 32 control the size of effective addresses and effective-address generation. When bits 31 and 32 are both zero, 24-bit addressing is specified. When bit 31 is zero and bit 32 is one, 31-bit addressing is specified. When bits 31 and 32 are both one, 64-bit addressing is specified. Bit 31 one and bit 32 zero is an invalid combination that causes a specification exception to be recognized. The addressing mode does not control the size of PER addresses or of addresses used to access DAT, ASN, dispatchable-unit-control, linkage, entry, and trace tables or access lists or the linkage stack. See “Address Generation” on page 5-6 and “Address Size and Wraparound” on page 3-5.
**Instruction Address:** Bits 64-127 of the PSW are the instruction address. This address designates the location of the leftmost byte of the next instruction to be executed, unless the CPU is in the wait state (bit 14 of the PSW is one).

Bit positions 0, 2-4, 24-30, and 33-63 are unsigned and must contain zeros. A specification exception is recognized when these bit positions do not contain zeros.

When bits 31 and 32 of the PSW specify the 24-bit addressing mode, bits 64-103 of the instruction address must be zeros, or, when bits 31 and 32 specify the 31-bit mode, bits 64-96 must be zeros. Otherwise, a specification exception is recognized. A specification exception is also recognized when bit 31 is one and bit 32 is zero or when bit position 12 does not contain a zero.

LOAD PSW EXTENDED has a 16-byte second operand. The instruction loads the operand unchanged and without examination as the current PSW.

LOAD PSW has an eight-byte second operand. The operand is treated as an ESA/390 PSW, except that bit 31 (the z/Architecture extended-addressing-mode bit) may be one.

Depending on the model, either LOAD PSW recognizes a specification exception if bit 12 of its second operand is not one or this error is indicated by an early specification exception after the completion of the execution of LOAD PSW. LOAD PSW loads bits 0-32 of its second operand, except with bit 12 inverted, and bits 33-63 of the operand as bits 0-32 and 97-127, respectively, of the current PSW, and it sets bits 33-96 of the current PSW to zeros.

**Control Registers**

The control registers provide for maintaining and manipulating control information outside the PSW. There are sixteen 64-bit control registers.

The LOAD CONTROL (LCTLG) instruction causes all control-register bit positions within those registers designated by the instruction to be loaded from storage. The LOAD CONTROL (LCTL) instruction loads only bit positions 32-63 of the control registers, and bits 0-31 of the registers remain unchanged. The instructions BRANCH AND SET AUTHORITY, BRANCH AND STACK, BRANCH IN SUBSPACE GROUP, EXTRACT AND SET EXTENDED AUTHORITY, LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE, provide specialized functions to place information into certain control-register bit positions.

Information loaded into the control registers becomes active (that is, assumes control over the system) at the completion of the instruction that causes the information to be loaded.

At the time the registers are loaded, the information is not checked for exceptions, such as an address designating an unavailable or protected location. The validity of the information is checked and the exceptions, if any, are indicated at the time the information is used.

The STORE CONTROL (STCTG) instruction causes the contents of all control-register bit positions, within those registers designated by the instruction, to be placed in storage. The STORE CONTROL (STCTL) instruction places the contents of bit positions 32-63 of the control registers in storage, and bits 0-31 of the registers are ignored. The instructions EXTRACT AND SET EXTENDED AUTHORITY, EXTRACT PRIMARY ASN, EXTRACT PRIMARY ASN AND INSTANCE, EXTRACT SECONDARY ASN, EXTRACT SECONDARY ASN AND INSTANCE, and PROGRAM CALL provide specialized functions to obtain information from certain control-register bit positions.
Only the general structure of the control registers is described here; the definition of a particular control-register bit position appears in the description of the facility with which the position is associated. Figure 4-4 shows the control-register bit positions which are assigned and the initial values of the positions upon execution of initial CPU reset. All control-register bit positions not listed in the figure are initialized to zero.

<table>
<thead>
<tr>
<th>Ctrl Reg</th>
<th>Bits</th>
<th>Name of Field</th>
<th>Associated with</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>33</td>
<td>SSM-suppression control</td>
<td>SET SYSTEM MASK</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>34</td>
<td>TOD-clock-sync control</td>
<td>TOD clock</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>35</td>
<td>Low-address-protection control</td>
<td>Low-address protection</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>36</td>
<td>Extraction-authority control</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>37</td>
<td>Secondary-space control</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>38</td>
<td>Fetch-protection-override control</td>
<td>Key-controlled protection</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>39</td>
<td>Storage-protection-override control</td>
<td>Key-controlled protection</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>44</td>
<td>ASN-and-LX-reuse control</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>45</td>
<td>AFP-register control</td>
<td>Floating point</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>Malfunction-alert subclass mask</td>
<td>External interruptions</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>49</td>
<td>Emergency-signal subclass mask</td>
<td>External interruptions</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>50</td>
<td>External-call subclass mask</td>
<td>External interruptions</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>52</td>
<td>Clock-comparator subclass mask</td>
<td>External interruptions</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>53</td>
<td>CPU-timer subclass mask</td>
<td>External interruptions</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>54</td>
<td>Service-signal subclass mask</td>
<td>External interruptions</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>56</td>
<td>Unused⁴</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>57</td>
<td>Interrupt-key subclass mask</td>
<td>External interruptions</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>58</td>
<td>Unused⁴</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>59</td>
<td>ETR subclass mask</td>
<td>External interruptions</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>61</td>
<td>Crypto control</td>
<td>Cryptography</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0-51</td>
<td>Primary region-table origin²</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0-51</td>
<td>Primary segment-table origin²</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0-51</td>
<td>Primary real-space token origin²</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>54</td>
<td>Primary subspace-group control</td>
<td>Subspace groups</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>55</td>
<td>Primary private-space control</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>56</td>
<td>Primary storage-alteration-event control</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>57</td>
<td>Primary space-switch-event control</td>
<td>Program interruptions</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>58</td>
<td>Primary real-space control</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>60-61</td>
<td>Primary designation-type control³</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>62-63</td>
<td>Primary table length³</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>33-57</td>
<td>Dispatchable-unit-control-table origin</td>
<td>Access-register translation</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0-31</td>
<td>Secondary ASN-second-table-entry instance number</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>32-47</td>
<td>PSW-key mask</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>48-63</td>
<td>Secondary ASN</td>
<td>Address spaces</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4-4 (Part 1 of 3). Assignment of Control-Register Fields

Programming Notes:

1. The detailed definition of a particular control-register bit position can be located by referring to the entry “control-register assignment” in the Index.

2. To ensure that existing programs operate correctly if and when new facilities using additional control-register bit positions are installed, the program should load zeros in unassigned positions.
<table>
<thead>
<tr>
<th>Ctrl Reg</th>
<th>Bits</th>
<th>Name of Field</th>
<th>Associated with</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0-31</td>
<td>Primary ASN-second-table-entry instance number</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>32-47</td>
<td>Authorization index</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>48-63</td>
<td>Primary ASN</td>
<td>Address spaces</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>33-57</td>
<td>Primary-ASN-second-table-entry origin</td>
<td>Access-register translation</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>32-39</td>
<td>I/O-interruption subclass mask</td>
<td>I/O interruptions</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0-51</td>
<td>Secondary segment-table origin(^2)</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0-51</td>
<td>Secondary region-table origin(^2)</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0-51</td>
<td>Secondary real-space token origin(^2)</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>54</td>
<td>Secondary subspace-group control</td>
<td>Subspace groups</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>Secondary private-space control</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>56</td>
<td>Secondary storage-alteration-event control</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>58</td>
<td>Secondary real-space control</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>60-61</td>
<td>Secondary designation-type control(^3)</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>62-63</td>
<td>Secondary table length(^3)</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>32-47</td>
<td>Extended authorization index</td>
<td>Access-register translation</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>48-63</td>
<td>Monitor masks</td>
<td>MONITOR CALL</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>32</td>
<td>Successful-branching-event mask</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>33</td>
<td>Instruction-fetching-event mask</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>Storage-alteration-event mask</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>Store-using-real-address-event mask</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>40</td>
<td>Branch-address control</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
<td>Storage-alteration-space control</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0-63</td>
<td>PER starting address</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0-63</td>
<td>PER ending address</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>Branch-trace control</td>
<td>Tracing</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>Mode-trace control</td>
<td>Tracing</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>2-61</td>
<td>Trace-entry address</td>
<td>Tracing</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>62</td>
<td>ASN-trace control</td>
<td>Tracing</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>63</td>
<td>Explicit-trace control</td>
<td>Tracing</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4-4 (Part 2 of 3). Assignment of Control-Register Fields
<table>
<thead>
<tr>
<th>Ctrl Reg</th>
<th>Bits</th>
<th>Name of Field</th>
<th>Associated with</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0-51</td>
<td>Home segment-table origin²</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0-51</td>
<td>Home region-table origin²</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0-51</td>
<td>Home real-space token origin²</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>55</td>
<td>Home private-space control</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>56</td>
<td>Home storage-alteration-event control</td>
<td>Program-event recording</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>57</td>
<td>Home space-switch-event control</td>
<td>Program interruptions</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>58</td>
<td>Home real-space control</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>60-61</td>
<td>Home designation-type control³</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>62-63</td>
<td>Home table length³</td>
<td>Dynamic address translation</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>32</td>
<td>Unused¹</td>
<td>I/O machine-check handling</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>33</td>
<td>Unused¹</td>
<td>Machine-check handling</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>35</td>
<td>Channel-report-pending subclass mask</td>
<td>Machine-check handling</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>36</td>
<td>Recovery subclass mask</td>
<td>Machine-check handling</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>37</td>
<td>Degradation subclass mask</td>
<td>Machine-check handling</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>38</td>
<td>External-damage subclass mask</td>
<td>Machine-check handling</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>39</td>
<td>Warning subclass mask</td>
<td>Machine-check handling</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>42</td>
<td>TOD-clock-control-override control</td>
<td>TOD clock</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>44</td>
<td>ASN-translation control</td>
<td>Instruction authorization</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>45-63</td>
<td>ASN-first-table origin</td>
<td>ASN translation</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0-60</td>
<td>Linkage-stack-entry address</td>
<td>Linkage-stack operations</td>
<td>0</td>
</tr>
</tbody>
</table>

Explanations:

The fields not listed are unassigned. The initial value for all unlisted control-register bit positions is zero.

¹ This bit is not used but is initialized to one for consistency with the System/370 definition.

² The address-space-control element (ASCE) in the control register has one of three formats, depending on bit 58 of the register, the real-space control, and bits 60 and 61 of the register, the designation-type control. When bit 58 is zero, the ASCE is a region-table designation if bits 60 and 61 are 11, 10, or 01 binary, or it is a segment-table designation if bits 60 and 61 are 00 binary. When bit 58 is one, the ASCE is a real-space designation. Bits 0-51 are the region-table origin, the segment-table origin or the real-space token origin, depending on whether the ASCE is a region-table designation, a segment-table designation, or a real-space designation, respectively.

³ Bits 60-63 are assigned when the ASCE in the control register is a region-table designation or a segment-table designation.

Figure 4-4 (Part 3 of 3). Assignment of Control-Register Fields

Tracing

Tracing assists in the determination of system problems by providing an ongoing record in storage of significant events. Tracing consists of four separately controllable functions which cause entries to be made in a trace table: branch tracing, ASN tracing, mode tracing, and explicit tracing. Branch tracing, ASN tracing, and mode tracing together are referred to as implicit tracing.

When branch tracing is on, a branch trace entry is made in the trace table for each execution of certain branch instructions when they cause branching. The branch address is placed in the trace entry. The trace entry also indicates the following about the addressing mode in effect after
branching and the branch address: (1) the CPU is in the 24-bit addressing mode, (2) the CPU either is in the 31-bit addressing mode or is in the 64-bit addressing mode and bits 0-32 of the branch address are all zeros, or (3) the CPU is in the 64-bit addressing mode and bits 0-32 of the branch address are not all zeros. The branch instructions that are traced are:

- BRANCH AND LINK (BALR only) when the \(R_z\) field is not zero
- BRANCH AND SAVE (BASR only) when the \(R_z\) field is not zero
- BRANCH AND SAVE AND SET MODE when the \(R_z\) field is not zero
- BRANCH AND SET AUTHORITY
- BRANCH AND STACK when the \(R_z\) field is not zero
- BRANCH IN SUBSPACE GROUP
- RESUME PROGRAM
- TRAP

However, a branch trace entry is made for \(\text{BRANCH IN SUBSPACE GROUP}\) only if ASN tracing is not on.

If both branch tracing and mode tracing are on and \(\text{BRANCH AND SAVE AND SET MODE}\) or \(\text{RESUME PROGRAM}\) changes the extended-addressing-mode bit, PSW bit 31, a mode-switching-branch trace entry is made instead of a branch trace entry.

When ASN tracing is on, an entry named the same as the instruction is made in the trace table for each execution of the following instructions:

- \(\text{BRANCH IN SUBSPACE GROUP}\)
- \(\text{PROGRAM CALL}\)
- \(\text{PROGRAM RETURN}\)
- \(\text{PROGRAM TRANSFER}\)
- \(\text{SET SECONDARY ASN}\)

However, the entry for \(\text{PROGRAM RETURN}\) is made only when \(\text{PROGRAM RETURN}\) unstacks a linkage-stack state entry that was formed by \(\text{PROGRAM CALL}\), not when \(\text{PROGRAM RETURN}\) unstacks an entry formed by \(\text{BRANCH AND STACK}\).

| When ASN tracing is on, a \(\text{PROGRAM TRANSFER}\) trace entry is also made for each execution of \(\text{PROGRAM TRANSFER WITH INSTANCE}\), and a \(\text{SET SECONDARY ASN}\) trace entry is also made for each execution of \(\text{SET SECONDARY ASN WITH INSTANCE}\). In either case, a bit in the trace entry indicates whether the entry was made due to the without-instance or the with-instance instruction.

If both ASN tracing and mode tracing are on and \(\text{PROGRAM CALL}\) uses a 20-bit PC number and changes PSW bit 31, first a \(\text{PROGRAM CALL}\) trace entry is made, and then a mode-switch trace entry is made. In this case except when \(\text{PROGRAM CALL}\) uses a 32-bit PC number, only a \(\text{PROGRAM CALL}\) trace entry is made since it indicates the old and new values of the extended-addressing-mode bit, PSW bit 31. A 32-bit PC number may be used if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0.

Mode tracing records a switch from a basic (24-bit or 31-bit) addressing mode to the extended (64-bit) addressing mode or from the extended mode to a basic mode.

When mode tracing is on, a mode-switch trace entry is made in the trace table for each execution of the following instructions if the execution changes PSW bit 31:

- \(\text{BRANCH AND SAVE AND SET MODE}\)
- \(\text{BRANCH AND SET MODE}\)
- \(\text{PROGRAM CALL}\)
- \(\text{PROGRAM RETURN}\)
- \(\text{RESUME PROGRAM}\)
- \(\text{SET ADDRESSING MODE}\)

However, a mode-switch trace entry is not made for \(\text{PROGRAM RETURN}\) if ASN tracing is on and \(\text{PROGRAM RETURN}\) unstacks a state entry formed by \(\text{PROGRAM CALL}\); a \(\text{PROGRAM RETURN}\) trace entry is made instead, and it contains information about PSW bit 31.

\(\text{BRANCH AND SAVE AND SET MODE}\) and \(\text{RESUME PROGRAM}\) cause trace entries to be made as follows: a branch trace entry if only branch tracing is on, a mode-switching-branch trace entry if both branch tracing and mode tracing are on, or a mode-switch trace entry if only mode tracing is on.

The trace entries produced by implicit tracing are summarized in [Figure 4-5 on page 4-13](#).

When explicit tracing is on, execution of \(\text{TRACE}\) (\(\text{TRACE}\) or \(\text{TRACG}\)) causes an entry to be made.
in the trace table. The entry for TRACE (TRACE) includes bits 16-63 from the TOD clock, the second operand of the TRACE instruction, and bits 32-63 of a range of general registers. The entry for TRACE (TRACG) is the same except that it includes bits 0-79 from the TOD clock and bits 0-63 of a range of general registers.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Trace Entries Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAKR</td>
<td>B - - B B - B</td>
</tr>
<tr>
<td>BALR</td>
<td>B - - B B - B</td>
</tr>
<tr>
<td>BASR</td>
<td>B - - B B - B</td>
</tr>
<tr>
<td>BASSM</td>
<td>B - MS B B</td>
</tr>
<tr>
<td>BSA</td>
<td>B - - B B - B</td>
</tr>
<tr>
<td>BSG</td>
<td>B BSG - BSG B BSG</td>
</tr>
<tr>
<td>BSM</td>
<td>- - MS - MS - MS</td>
</tr>
<tr>
<td>PC-20</td>
<td>- PC MS PC MS PC &amp; MS PC &amp; MS</td>
</tr>
<tr>
<td>PC-32</td>
<td>- PC MS PC MS PC PC</td>
</tr>
<tr>
<td>PR-b</td>
<td>- - MS - MS MS MS</td>
</tr>
<tr>
<td>PR-pc</td>
<td>- PR MS PR MS PR PR</td>
</tr>
<tr>
<td>PT or</td>
<td>- PT - PT - PT PT</td>
</tr>
<tr>
<td>PTI</td>
<td></td>
</tr>
<tr>
<td>RP</td>
<td>B - MS B B</td>
</tr>
<tr>
<td>SSAR or</td>
<td>- SSAR - SSAR - SSAR SSAR</td>
</tr>
<tr>
<td>SSAIR</td>
<td></td>
</tr>
<tr>
<td>SAM24/31/64</td>
<td>- - MS - MS MS MS</td>
</tr>
<tr>
<td>TRAP2/4</td>
<td>B - - B B - B</td>
</tr>
</tbody>
</table>

**Explanation:**

- None.
- **-20** The case when PROGRAM CALL uses a 20-bit PC number.
- **-32** The case when PROGRAM CALL uses a 32-bit PC number.
- **-b** The case when PROGRAM RETURN unstacks a branch state entry.
- **-pc** The case when PROGRAM RETURN unstacks a program-call state entry.

*Figure 4-5 (Part 1 of 2). Summary of Implicit Tracing*
Explanation (Continued):

<table>
<thead>
<tr>
<th>OR.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND.</td>
</tr>
</tbody>
</table>

B Branch trace entry. Made only if the branch is taken and a mode-switching-branch trace entry is not made.

MS Mode-switch trace entry. Made only if PSW bit 31 is changed.

MSB Mode-switching-branch trace entry. Made only if PSW bit 31 is changed (which can occur only if the branch is taken).

Figure 4-5 (Part 2 of 2). Summary of Implicit Tracing

Control-Register Allocation

The information to control tracing is contained in control register 12 and has the following format:

<table>
<thead>
<tr>
<th>B</th>
<th>M</th>
<th>Trace-Entry Address</th>
<th>A</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

Branch-Trace-Control Bit (B): Bit 0 of control register 12 controls whether branch tracing is turned on or off. If the bit is zero, branch tracing is off; if the bit is one, branch tracing is on.

Mode-Trace-Control Bit (M): Bit 1 of control register 12 controls whether mode tracing is turned on or off. If the bit is zero, mode tracing is off; if the bit is one, mode tracing is on.

Trace-Entry Address: Bits 2-61 of control register 12, with two zero bits appended on the left and two on the right, form the real address of the next trace entry to be made.

ASN-Trace-Control Bit (A): Bit 62 of control register 12 controls whether ASN tracing is turned on or off. If the bit is zero, ASN tracing is off; if the bit is one, ASN tracing is on.

Explicit-Trace-Control Bit (E): Bit 63 of control register 12 controls whether explicit tracing is turned on or off. If the bit is zero, explicit tracing is off, which causes the TRACE instruction to be executed as a no-operation; if the bit is one, the execution of the TRACE instruction creates an entry in the trace table, except that no entry is made when bit 0 of the second operand of the TRACE instruction is one.

Trace Entries

Trace entries are of nine types, with most types having more than one detailed format. The types and numbers of formats are as follows:

- Branch (three formats)
- BRANCH IN SUBSPACE GROUP (two formats)
- Mode switch (three formats)
- Mode-switching branch (three formats)
- PROGRAM CALL (seven formats)
- PROGRAM RETURN (nine formats)
- PROGRAM TRANSFER (three formats)
- SET SECONDARY ASN (one format)
- TRACE (two formats)

Format-1 and format-2 PROGRAM CALL trace entries are made if the ASN-and-LX reuse facility is not enabled. Entries of formats 3-7 are made if the facility is enabled.

The PROGRAM TRANSFER trace entry is also made for PROGRAM TRANSFER WITH INSTANCE, and the SET SECONDARY ASN trace entry is also made for SET SECONDARY ASN WITH INSTANCE. In either case, bit 15 (N) of the entry is one if the entry was made because of execution of the with-instance instruction.
The entries are shown in Figure 4-6 on page 4-15. In that figure, each entry is labeled with “Fn,” indicating a format number, to allow references to each format within a trace-entry type. Also, “Branch,” referring to the mnemonic of an instruction that causes a branch trace entry, refers to BAKR, BALR, BASR, BASSM, BSA, or BSG.

Figure 4-7 on page 4-23 lists the trace entries in ascending order of values in bit fields that identify the entries.

F1 Branch (Branch, RP, or TRAP2/4 when Resulting Mode Is 24-Bit)

00000000 Bits 40-63 of Branch Adr.
0 8 31

F2 Branch (Branch, RP, or TRAP2/4 when Resulting Mode Is 31-Bit, or when Resulting PSW Bit 31 Is One (See Note) and Bits 0-32 of Branch Address Are All Zeros)

1 Bits 33-63 of Branch Address
0 1 31

F3 Branch (Branch, RP, or TRAP2/4 when Resulting PSW Bit 31 Is One (See Note) and Bits 0-32 of Branch Address Are Not All Zeros)

01010010 All Zeros Bits 0-31 of Branch Address
0 8 12 32 63

F1 BRANCH IN SUBSPACE GROUP (if ASN Is Tracing on, in 24-Bit or 31-Bit Mode)

01000001 Bits 9-31 of ALET A Bits 33-63 of Branch Address
0 8 32 63

F2 BRANCH IN SUBSPACE GROUP (if ASN Is Tracing on, in 64-Bit Mode)

01000010 Bits 9-31 of ALET Bits 0-31 of Branch Address
0 8 32 63

F1 Mode Switch (BASSM, BSM, PC, PR, RP, or SAM64 from 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note))

010100010011 All Zeros A Updated Instruction Address
0 8 12 32 63

Figure 4-6 (Part 1 of 9). Trace Entries
F2 Mode Switch (BASSM, BSM, PC, PR, RP, SAM24, or SAM31 from 64-Bit Mode to 24-Bit or 31-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros)

<table>
<thead>
<tr>
<th>01010001</th>
<th>0010</th>
<th>All Zeros</th>
<th>Bits 32-63 of Updated Inst. Adr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>32</td>
</tr>
</tbody>
</table>

F3 Mode Switch (BASSM, BSM, PC, PR, RP, SAM24, or SAM31 from 64-Bit Mode to 24-Bit or 31-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros)

<table>
<thead>
<tr>
<th>01010010</th>
<th>0110</th>
<th>All Zeros</th>
<th>Bits 0-31 of Updated Inst. Adr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>32</td>
</tr>
</tbody>
</table>

Bits 32-63 of Updated Inst. Adr.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>95</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F1 Mode-Switching Branch (BASSM or RP from 64-Bit Mode to 24-Bit or 31-Bit Mode)

<table>
<thead>
<tr>
<th>01010001</th>
<th>1010</th>
<th>All Zeros</th>
<th>A</th>
<th>Branch Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>32</td>
<td>63</td>
</tr>
</tbody>
</table>

F2 Mode-Switching Branch (BASSM or RP from 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Branch Address Are All Zeros)

<table>
<thead>
<tr>
<th>01010001</th>
<th>1011</th>
<th>All Zeros</th>
<th>Bits 32-63 of Branch Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>32</td>
</tr>
</tbody>
</table>

F3 Mode-Switching Branch (BASSM or RP from 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Branch Address Are Not All Zeros)

<table>
<thead>
<tr>
<th>01010010</th>
<th>1111</th>
<th>All Zeros</th>
<th>Bits 0-31 of Branch Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>32</td>
</tr>
</tbody>
</table>

Bits 32-63 of Branch Address

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-6 (Part 2 of 9). Trace Entries
F1 PROGRAM CALL (in 24-Bit or 31-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Not Enabled)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>PC Number</th>
<th>Bits 33-62 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100001</td>
<td>8 12 32</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

F2 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Not Enabled)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>PC Number</th>
<th>Bits /zerodot-31 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100010</td>
<td>8 12 32</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 32-62 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>95</td>
</tr>
</tbody>
</table>

F3 PROGRAM CALL (in 24-Bit or 31-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled and 2/zerodot-Bit PC Number Is Used)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>Bits 1-19 of 20-Bit PC Number</th>
<th>PSW Key</th>
<th>Bits 33-62 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100001</td>
<td>8 12 32</td>
<td>0</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

F4 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled and 2/zerodot-Bit PC Number Is Used)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>Bits 1-19 of 20-Bit PC Number</th>
<th>Bits /zerodot-31 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100010</td>
<td>8 12 32</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 32-62 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>95</td>
</tr>
</tbody>
</table>

Figure 4-6 (Part 3 of 9). Trace Entries
F5 PROGRAM CALL (in 24-Bit or 31-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled and 32-Bit PC Number Is Used)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>101E</th>
<th>All Zeros</th>
<th>Bits 33-62 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100010</td>
<td>100E</td>
<td>All Zeros</td>
<td>Bits 33-62 of Return Address</td>
</tr>
<tr>
<td>0 8 12 16 32</td>
<td>63</td>
<td>64</td>
<td>95</td>
</tr>
</tbody>
</table>

32-Bit PC Number

F6 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled, 32-Bit PC Number Is Used, and Bits 0-31 of Return Address Are All Zeros)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>101E</th>
<th>All Zeros</th>
<th>Bits 32-62 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100010</td>
<td>101E</td>
<td>All Zeros</td>
<td>Bits 32-62 of Return Address</td>
</tr>
<tr>
<td>0 8 12 16 32</td>
<td>63</td>
<td>64</td>
<td>95</td>
</tr>
</tbody>
</table>

32-Bit PC Number

F7 PROGRAM CALL (in 64-Bit Mode, Regardless of Resulting Mode, if ASN-and-LX Reuse Is Enabled, 32-Bit PC Number Is Used, and Bits 0-31 of Return Address Are Not All Zeros)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>111E</th>
<th>All Zeros</th>
<th>Bits 0-31 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100011</td>
<td>111E</td>
<td>All Zeros</td>
<td>Bits 0-31 of Return Address</td>
</tr>
<tr>
<td>0 8 12 16 32</td>
<td>63</td>
<td>64</td>
<td>96</td>
</tr>
</tbody>
</table>

Bits 32-62 of Return Address | 64 96 127 32-Bit PC Number

F1 PROGRAM RETURN (in 24-Bit or 31-Bit Mode when Resulting Mode Is 24-Bit or 31-Bit)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>0000</th>
<th>New PASN</th>
<th>Bits 33-62 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110010</td>
<td>0000</td>
<td>New PASN</td>
<td>Bits 33-62 of Return Address</td>
</tr>
<tr>
<td>0 8 12 16 32</td>
<td>63</td>
<td>64</td>
<td>95</td>
</tr>
</tbody>
</table>

A Bits 33-63 of Updated Inst. Adr.
F2 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros and Resulting Mode Is 24-Bit or 31-Bit)

<table>
<thead>
<tr>
<th>00110010</th>
<th>PSW</th>
<th>Key</th>
<th>0010</th>
<th>New PASN</th>
<th>A</th>
<th>Bits 33-62 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>32</td>
<td>63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits 32-63 of Updated Inst. Adr.

64 95

F3 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros and Resulting Mode Is 24-Bit or 31-Bit)

<table>
<thead>
<tr>
<th>00110011</th>
<th>PSW</th>
<th>Key</th>
<th>0011</th>
<th>New PASN</th>
<th>A</th>
<th>Bits 33-62 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>32</td>
<td>63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Updated Instruction Address

64 127

F4 PROGRAM RETURN (in 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Return Address Are All Zeros)

<table>
<thead>
<tr>
<th>00110010</th>
<th>PSW</th>
<th>Key</th>
<th>1000</th>
<th>New PASN</th>
<th>Bits 32-63 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>32</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

A Bits 33-63 of Updated Inst. Adr.

64 95

F5 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are All Zeros)

<table>
<thead>
<tr>
<th>00110010</th>
<th>PSW</th>
<th>Key</th>
<th>1010</th>
<th>New PASN</th>
<th>Bits 32-63 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>32</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

Bits 32-63 of Updated Inst. Adr.

64 95

Figure 4-6 (Part 5 of 9). Trace Entries
F6 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are All Zeros)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>New PASN</th>
<th>Bits 32-62 of Return Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110011</td>
<td>1011</td>
<td>32</td>
<td>63</td>
</tr>
</tbody>
</table>

Updated Instruction Address

64 127

F7 PROGRAM RETURN (in 24-Bit or 31-Bit Mode when Resulting PSW Bit 31 Is One (See Note) and Bits 0-31 of Return Address Are Not All Zeros)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>New PASN</th>
<th>Bits 0-31 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110011</td>
<td>1100</td>
<td>32</td>
</tr>
</tbody>
</table>

Bits 32-62 of Return Address  P  A  Updated Instruction Address

64 96 127

F8 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are Not All Zeros)

<table>
<thead>
<tr>
<th>PSW Key</th>
<th>New PASN</th>
<th>Bits 0-31 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110011</td>
<td>1110</td>
<td>32</td>
</tr>
</tbody>
</table>


64 96 127

Figure 4-6 (Part 6 of 9). Trace Entries
F9 PROGRAM RETURN (in 64-Bit Mode when Bits 0-31 of Updated Instruction Address Are Not All Zeros, Resulting PSW Bit 31 Is One (See Note), and Bits 0-31 of Return Address Are Not All Zeros)

<table>
<thead>
<tr>
<th>PSW</th>
<th>Key</th>
<th>New PASN</th>
<th>Bits 0-31 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110100</td>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>96</td>
<td>127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 32-63 of Updated Inst. Adr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
</tr>
</tbody>
</table>

F1 PROGRAM TRANSFER (WITH INSTANCE if N Is One) (in 24-Bit or 31-Bit Mode)

<table>
<thead>
<tr>
<th>PSW</th>
<th>Key</th>
<th>New PASN</th>
<th>Bits 32-63 of R before</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110001</td>
<td>000N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 32-63 of R before</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

F2 PROGRAM TRANSFER (WITH INSTANCE if N Is One) (in 64-Bit Mode when Bits 0-31 of R Are All Zeros)

<table>
<thead>
<tr>
<th>PSW</th>
<th>Key</th>
<th>New PASN</th>
<th>Bits 32-63 of R before</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110001</td>
<td>100N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 32-63 of R before</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

F3 PROGRAM TRANSFER (WITH INSTANCE if N Is One) (in 64-Bit Mode when Bits 0-31 of R Are Not All Zeros)

<table>
<thead>
<tr>
<th>PSW</th>
<th>Key</th>
<th>New PASN</th>
<th>Bits 0-31 of R before</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110010</td>
<td>110N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 32-63 of R before</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

Figure 4-6 (Part 7 of 9). Trace Entries
F1 SET SECONDARY ASN (WITH INSTANCE if N Is One)

<table>
<thead>
<tr>
<th>00010000</th>
<th>00000000N</th>
<th>New SASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

F1 TRACE (TRACE)

<table>
<thead>
<tr>
<th>0111 N 00000000</th>
<th>TOD-Clock Bits 16-63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 4 8 16 63</td>
<td></td>
</tr>
</tbody>
</table>

TRACE Operand | (R1) - (R3) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64 96</td>
<td>95 + 32(N+1)</td>
</tr>
</tbody>
</table>

F2 TRACE (TRACG)

<table>
<thead>
<tr>
<th>0111 N 10000000</th>
<th>TOD-Clock Bits 0-47</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 4 8 16 63</td>
<td></td>
</tr>
</tbody>
</table>

TOD-Clock Bits 48-79 | TRACE Operand |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64 96</td>
<td>127</td>
</tr>
</tbody>
</table>

(R1) - (R3) |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
</tr>
</tbody>
</table>

Figure 4-6 (Part 8 of 9). Trace Entries

Note: The terminology “when Resulting PSW Bit 31 Is One” is used instead of “when Resulting Mode Is 64-Bit” because, if the resulting PSW bit 32 is zero, an early specification exception will be recognized. PROGRAM RETURN can set PSW bit 31 to one and bit 32 to zero.

Figure 4-6 (Part 9 of 9). Trace Entries
<table>
<thead>
<tr>
<th>Trace-Entry Bits</th>
<th>Trace Entry</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7 8-11 12-15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>Branch</td>
<td>1</td>
</tr>
<tr>
<td>00001000</td>
<td>SET SECONDARY ASN</td>
<td>1</td>
</tr>
<tr>
<td>00100001</td>
<td>PROGRAM CALL</td>
<td>1\textsuperscript{1}</td>
</tr>
<tr>
<td>00100010</td>
<td>PROGRAM CALL</td>
<td>2\textsuperscript{1}</td>
</tr>
<tr>
<td>00100011</td>
<td>PROGRAM CALL</td>
<td>3\textsuperscript{1}</td>
</tr>
<tr>
<td>01000010</td>
<td>PROGRAM CALL</td>
<td>4\textsuperscript{1}</td>
</tr>
<tr>
<td>01000011</td>
<td>PROGRAM CALL</td>
<td>5\textsuperscript{1}</td>
</tr>
<tr>
<td>01000111</td>
<td>PROGRAM CALL</td>
<td>6\textsuperscript{1}</td>
</tr>
<tr>
<td>01010011</td>
<td>PROGRAM CALL</td>
<td>7\textsuperscript{1}</td>
</tr>
<tr>
<td>00110001</td>
<td>PROGRAM TRANSFER</td>
<td>1</td>
</tr>
<tr>
<td>00110010</td>
<td>PROGRAM TRANSFER</td>
<td>2</td>
</tr>
<tr>
<td>00110011</td>
<td>PROGRAM TRANSFER</td>
<td>3</td>
</tr>
<tr>
<td>01000011</td>
<td>PROGRAM RETURN</td>
<td>1</td>
</tr>
<tr>
<td>01000011</td>
<td>PROGRAM RETURN</td>
<td>2</td>
</tr>
<tr>
<td>01000101</td>
<td>PROGRAM RETURN</td>
<td>3</td>
</tr>
<tr>
<td>01000111</td>
<td>PROGRAM RETURN</td>
<td>4</td>
</tr>
<tr>
<td>01010011</td>
<td>PROGRAM RETURN</td>
<td>5</td>
</tr>
<tr>
<td>01010012</td>
<td>PROGRAM RETURN</td>
<td>6</td>
</tr>
<tr>
<td>01010013</td>
<td>PROGRAM RETURN</td>
<td>7</td>
</tr>
<tr>
<td>01010014</td>
<td>PROGRAM RETURN</td>
<td>8</td>
</tr>
<tr>
<td>01101001</td>
<td>PROGRAM RETURN</td>
<td>9</td>
</tr>
<tr>
<td>01101010</td>
<td>BRANCH IN SUBSPACE GROUP</td>
<td>1</td>
</tr>
<tr>
<td>01101011</td>
<td>BRANCH IN SUBSPACE GROUP</td>
<td>2</td>
</tr>
<tr>
<td>01101101</td>
<td>MODE SWITCH</td>
<td>2</td>
</tr>
<tr>
<td>01101111</td>
<td>MODE SWITCH</td>
<td>1</td>
</tr>
<tr>
<td>01110010</td>
<td>MODE-Switching Branch</td>
<td>1</td>
</tr>
<tr>
<td>01110011</td>
<td>MODE-Switching Branch</td>
<td>2</td>
</tr>
<tr>
<td>01110101</td>
<td>MODE-Switching Branch</td>
<td>3</td>
</tr>
<tr>
<td>01110111</td>
<td>MODE-Switching Branch</td>
<td>3</td>
</tr>
<tr>
<td>01111001</td>
<td>TRACE</td>
<td>1</td>
</tr>
<tr>
<td>01111010</td>
<td>TRACE</td>
<td>2</td>
</tr>
<tr>
<td>01111110</td>
<td>TRACE</td>
<td>2</td>
</tr>
<tr>
<td>1001111000</td>
<td>BRANCH</td>
<td>2</td>
</tr>
<tr>
<td>1001111010</td>
<td>BRANCH</td>
<td>3</td>
</tr>
<tr>
<td>11110111</td>
<td>BRANCH</td>
<td>3</td>
</tr>
</tbody>
</table>

*Figure 4-7 (Part 1 of 2). Trace Entries Arranged by Identifying Bits*
Explanation:

1 Format-1 and format-2 entries are made when the ASN-and-LX-reuse facility is not enabled. Entries of formats 3-7 are made when the facility is enabled.

E Indicates, when one, that the extended-addressing-mode bit, PSW bit 31, was set to one.

N Indicates, when one, that a PROGRAM TRANSFER entry was made because of PROGRAM TRANSFER WITH INSTANCE or a SET SECONDARY ASN entry was made because of SET SECONDARY ASN WITH INSTANCE.

Figure 4-7 (Part 2 of 2). Trace Entries Arranged by Identifying Bits

The fields in the trace entries are defined as follows. The fields are described in the order in which they first appear in Figure 4-6 on page 4-15.

Branch Address: The branch address is the address of the next instruction to be executed when the branch is taken. In a branch trace entry made when the 24-bit addressing mode is in effect after branching (a format-1 entry), bit positions 8-31 contain bits 40-63 of the branch address. When the 31-bit addressing mode is in effect after branching or PSW bit 31 is one after branching and bits 0-32 of the branch address are all zeros, bit positions 1-31 of the trace entry (format 2) contain bits 33-63 of the branch address. When PSW bit 31 is one after branching and bits 0-32 of the branch address are not all zeros, bit positions 32-95 of the trace entry (format 3), contain bits 0-63 of the branch address.

In a BRANCH IN SUBSPACE GROUP trace entry made on execution in the 24-bit or 31-bit addressing mode, bit positions 33-63 of the trace entry (format 1) contain bits 33-63 of the branch address, or, in the 64-bit addressing mode, bit positions 32-95 of the trace entry (format 2) contain bits 0-63 of the branch address.

In a mode-switching-branch trace entry made on a switch from the 64-bit addressing mode to the 24-bit or 31-bit addressing mode, bit positions 33-63 of the entry (format 1) contain bits 33-63 of the branch address; or, on a switch from PSW bit 31 being off to the bit being on, bit positions 32-63 of the entry (format 2) contain bits 32-63 of the branch address if bits 0-31 of the branch address are zeros, or bits 32-95 of the entry (format 3) contain bits 0-63 of the branch address if bits 0-31 of the branch address are not all zeros.

Primary-List Bit (P) and Bits 9-31 of ALET: Bit position 8 of a BRANCH IN SUBSPACE GROUP trace entry contains bit 7 of the access-list-entry token (ALET) in the access register designated by the Rbasesupertwo field of the instruction. Bit positions 9-31 of the trace entry contain bits 9-31 of the ALET.

Basic-Addressing-Mode Bit (A): Bit position 32 of a BRANCH IN SUBSPACE GROUP trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1 entry) contains the basic-addressing-mode bit that replaces bit 32 of the PSW.

Bit position 32 of a mode-switch trace entry that indicates a switch from PSW bit 31 being off to the bit being on (a format-1 entry) contains the value of PSW bit 32 that existed before the mode-switch operation.

Bit position 32 of a mode-switching-branch trace entry that indicates a switch from the 64-bit addressing mode to the 24-bit or 31-bit addressing mode (a format-1 entry) contains the value that replaces PSW bit 32.

Bit position 32 of a PROGRAM CALL trace entry made on execution in the 24-bit or 31-bit addressing mode (regardless of the resulting addressing mode) (a format-1 entry) contains the basic-addressing-mode bit, bit 32, from the current PSW.
Bit position 32 of a PROGRAM RETURN trace entry made when the resulting addressing mode is the 24-bit or 31-bit mode (a format-1, format-2, or format-3 entry) contains the basic-addressing-mode bit that replaces bit 32 of the PSW.

Bit position 64 of a PROGRAM RETURN trace entry made in the 24-bit or 31-bit addressing mode when the return address occupies only one word in the entry, (a format-1 or format-4 entry), contains the value of PSW bit 32 that existed before the PROGRAM RETURN operation. When the return address occupies two words (a format-7 entry), bit position 96 contains that value of PSW bit 32.

**Updated Instruction Address:** Bit positions 33-63 of a mode-switch trace entry that indicates a switch from PSW bit 31 being off to the bit being on (a format-1 entry) contains bits 33-63 of the updated instruction address in the PSW (bits 97-127 of the PSW) before that address is replaced, if it is replaced, by the mode-switch operation. Bit positions 32-63 of a mode-switch trace entry (format 2) that indicates a switch from the 64-bit addressing mode to the 24-bit or 31-bit addressing mode contains bits 32-63 of the updated instruction address in the PSW (bits 96-127 of the PSW) before that address is replaced, if it is replaced, by the mode-switch operation, if bits 0-31 of the updated instruction address are zeros; or bit positions 32-95 of the trace entry (format 3) contain bits 0-63 of that updated instruction address (bits 64-127 of the PSW) if bits 0-31 of the address are not all zeros.

The following description of a PROGRAM RETURN trace entry applies when the return address in the entry occupies only one word in the entry. Bit positions 65-95 of the trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1 or format-4 entry) contain bits 33-63 of the updated instruction address in the PSW (bits 97-127 of the PSW) before that address is replaced from the linkage-stack state entry; or, when the execution is in the 64-bit addressing mode, bit positions 32-94 of the trace entry (format 2, 4, or 7) contain bits 0-62 of that updated instruction address (bits 64-126 of the PSW), or, when bits 0-31 of the address are all zeros, bit positions 32-62 of the trace entry (format 6) contain bits 32-62 of the address.

**Return Address:** Bit positions 33-62 of a PROGRAM CALL trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1, format-3, or format-5 entry) contain bits 33-62 of the updated instruction address in the PSW (bits 97-127 of the PSW) before that address is replaced from the entry-table entry; or, when the execution is in the 64-bit addressing mode, bit positions 32-94 of the trace entry (format 2, 4, or 7) contain bits 0-62 of that updated instruction address (bits 64-126 of the PSW), or, when bits 0-31 of the address are all zeros, bit positions 32-62 of the trace entry (format 6) contain bits 32-62 of the address.

**Extended-Addressing-Mode Bit (E):** Bit position 15 of a PROGRAM CALL trace entry made using a 32-bit PC number (a format-5, format-6, or format-7 entry) contains the extended-addressing-mode bit that replaces bit 31 of the PSW.

Bit positions 33-62 of a PROGRAM RETURN trace entry made when the resulting addressing mode is the 24-bit or 31-bit mode (a format-1, format-2, or format-3 entry) contain bits 33-62 of the instruction address that replaces bits 64-127 of the PSW; or, when the resulting PSW bit 31 is one (which causes the addressing mode be the 64-bit mode unless the resulting PSW bit 32 is zero), bit positions 32-62 of the trace entry (formats 4-6) contain bits 32-62 of that instruction address if bits 0-31 of the address are zeros, or bit positions 32-94 of the trace entry (formats 7-9) contain bits 0-63 of that updated instruction address (bits 64-127 of the PSW) if bits 0-31 of the address are not all zeros. If the return address in the PROGRAM RETURN trace entry occupies two words, the updated instruction address in the entry is moved one word to the right in the entry (formats 7-9).

**PSW Key:** Bit positions 8-11 of a PROGRAM CALL, PROGRAM TRANSFER, or PROGRAM RETURN trace entry contain the PSW key from the current PSW.

**PC Number:** Bit positions 12-31 of a PROGRAM CALL trace entry of format 1-4 contain the value of the rightmost 20 bits of the second-operand address. Bit positions 64-95 of a format-5 or format-6 PROGRAM CALL trace entry, or bit positions 96-127 of a format-7 entry, contain the value of the rightmost 32 bits of the second-operand address.
contain bits 0-62 of that instruction address if bits 0-31 of the address are not all zeros.

**Problem-State Bit (P):** Bit position 63 of a PROGRAM CALL trace entry made on execution in the 24-bit or 31-bit addressing mode (regardless of the resulting mode) (a format-1, format-3, format-5, or format-6 entry), or bit position 95 of the entry (format 2, 4, or 7) made on execution in the 64-bit addressing mode, contains the problem-state bit from the current PSW.

Bit position 63 of a PROGRAM RETURN trace entry made when the resulting addressing mode is the 24-bit or 31-bit mode (a format-1, format-2, or format-3 entry) or when the resulting PSW bit 31 is one and bits 0-31 of the return address are zeros (formats 4-6) contains the problem-state bit that replaces bit 15 of the PSW. Bit position 95 of a PROGRAM RETURN trace entry made when the returning PSW bit 31 is one and bits 0-31 of the return address are not all zeros (formats 7-9) contains the problem-state bit.

**New PASN:** Bit positions 16-31 of a PROGRAM TRANSFER trace entry contain the new PASN (which may be zero) specified in bit positions 48-63 of general register $R_{\text{basesuperone}}$.

Bit positions 16-31 of a PROGRAM RETURN trace entry contain the new PASN that is restored from the linkage-stack state entry.

**Bits 32-63 of $R_z$ before:** Bit positions 32-63 of a PROGRAM TRANSFER trace entry made on execution in the 24-bit or 31-bit addressing mode (a format-1 entry) contain bits 32-63 of the general register designated by the $R_z$ field of the instruction. (Bits 32 and 33-62 of that register replace bits 32 and 97-126, respectively, of the PSW. Bit 63 of the register replaces the problem-state bit in the PSW.) When PROGRAM TRANSFER or PROGRAM TRANSFER WITH INSTANCE is executed in the 64-bit addressing mode, bit positions 32-63 of the trace entry (format 2) contain bits 32-63 of the $R_z$ general register if bits 0-31 of the register are zeros, or bit positions 32-95 of the trace entry (format 3) contain bits 0-63 of the register if bits 0-31 of the register are not all zeros.

**New SASN:** Bit positions 16-31 of a SET SECONDARY ASN trace entry contain the ASN value loaded into control register 3 by the instruction.

**Number of Registers (N):** Bits 4-7 of the trace entry for TRACE contain a value which is one less than the number of general registers which have been provided in the trace entry. The value of N ranges from zero, meaning the contents of one general register are provided in the trace entry, to 15, meaning the contents of all 16 general registers are provided.

**TOD-Clock Bits 16-63 or 0-79:** Bits 16-63 of the trace entry for TRACE (TRACE) are obtained from bit positions 16-63 of the TOD clock, as would be provided by a STORE CLOCK instruction executed at the time the TRACE instruction was executed. Bits 16-95 of the trace entry for TRACE (TRACG) are obtained from bit positions 0-79 of the TOD clock, as would be provided by a STORE CLOCK EXTENDED instruction executed at the time the TRACE instruction was executed. See programming note 2 for information about a carry from bit position 0 of the TOD clock.

**TRACE Operand:** Bit positions 64-95 of the trace entry for TRACE (TRACE) contain a copy of the 32 bits of the second operand of the TRACE instruction for which the entry is made. Bit positions 96-127 of the trace entry for TRACE (TRACG) contain a copy of those bits.

**(R_{\text{basesuperone}})-(R_{\text{basesuperthree}}):** The four-byte fields starting with bit 96 of the trace entry for TRACE (TRACE) contain the contents of bit positions 32-63 of the general registers whose range is specified by the $R_{\text{basesuperone}}$ and $R_{\text{basesuperthree}}$ fields of the TRACE instruction. The general registers are stored in ascending order of register numbers, starting with general register $R_{\text{basesuperone}}$ and continuing up to and including general register $R_{\text{basesuperthree}}$, with general register 0 following general register 15. The eight-byte fields starting with bit 128 of the trace entry for TRACE (TRACG) similarly contain the contents of bit positions 0-63 of those registers.

**Programming Notes:**

1. The size of the trace entry for TRACE (TRACE) in units of words is $3 + (N + 1)$. The maximum size of an entry is 19 words, or 76 bytes. For TRACE (TRACG), the size in units of words is $4 + 2(N + 1)$, and the maximum size is 36 words, or 144 bytes.

2. At some time in the future, the TOD clock on new models will have a leftmost extension so that there can be a carry from bit position 0 of
the clock into the extension; see programming note 14 on page 4-44. On these models, the rightmost bit of the extension will be stored in bit position 15 of the TRACE (TRACG) trace entry. It may be desired to have programs that process TRACE (TRACG) trace entries take this future development into account.

**Operation**

When an instruction which is subject to tracing is executed and the corresponding tracing function is turned on, a trace entry of the appropriate type and format is made. The real address of the trace entry is formed by appending two zero bits on the left and two on the right to the value in bit positions 2-61 of control register 12. The address in control register 12 is subsequently increased by the size of the entry created.

No trace entry is stored if the incrementing of the address in control register 12 would cause a carry to be propagated into bit position 51 (that is, if the trace-entry address would be in the next 4K-byte block). If this would be the case for the entry to be made, a trace-table exception is recognized. When PROGRAM CALL is to form both a PROGRAM CALL trace entry and a mode-switch trace entry, neither entry is stored, and a trace-table exception is recognized, if either entry would cause a carry into bit position 51. For the purpose of recognizing the trace-table exception in the case of a TRACE instruction, the maximum length of 76 (TRACE) or 144 (TRACG) bytes is used instead of the actual length.

The storing of a trace entry is not subject to key-controlled protection (nor, since the trace-entry address is real, is it subject to page protection), but it is subject to low-address protection; that is, if the address of the trace entry due to be created is in the range 0-511 or 4096-4607 and bit 35 of control register 0 is one, a protection exception is recognized, and instruction execution is suppressed. If the address of a trace entry is invalid, an addressing exception is recognized, and instruction execution is suppressed.

The three exceptions associated with storing a trace entry (addressing, protection, and trace table) are collectively referred to as trace exceptions.

If a program interruption takes place for a condition which is not a trace-exception condition and for which execution of an instruction is not completed, it is unpredictable whether part or all of any trace entry due to be made for such an interrupted instruction is stored in the trace table. Thus, for a condition which would ordinarily cause nullification or suppression of instruction execution, storage locations may have been altered beginning at the location designated by control register 12 and extending up to the length of the entry that would have been created.

When PROGRAM RETURN unstacks a linkage-stack state entry that was formed by BRANCH AND STACK and ASN tracing is on, trace exceptions may be recognized, even though a trace entry is not made and no part of a trace entry is stored.

The order in which information is placed in a trace entry is unpredictable. Furthermore, as observed by other CPUs and by channel programs, the contents of a byte of a trace entry may appear to change more than once before completion of the instruction for which the entry is made.

The trace-entry address in control register 12 is updated only on completion of execution of an instruction for which a trace entry is made.

A serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. However, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.

**Program-Event Recording**

The purpose of PER is to assist in debugging programs. It permits the program to be alerted to the following types of events:

- Execution of a successful branch instruction. The option is provided of having an event occur only when the branch-target location is within the designated storage area.
- Fetching of an instruction from the designated storage area.
- Alteration of the contents of the designated storage area. The option is provided of
having an event occur only when the storage area is within designated address spaces.

- Execution of the STORE USING REAL ADDRESS instruction.

The program can selectively specify that one or more of the above types of events be recognized, except that the event for STORE USING REAL ADDRESS can be specified only along with the storage-alteration event. The information concerning a PER event is provided to the program by means of a program interruption, with the cause of the interruption being identified in the interruption code.

Control-Register Allocation and Address-Space-Control Element

The information for controlling PER resides in control registers 9, 10, and 11 and the address-space-control element. The information in the control registers has the following format:

Control Register 9

```
  EM  B  S
0  32 37 40 42 63
```

Control Register 10

```
 Starting Address
0  63
```

Control Register 11

```
 Ending Address
0  63
```

**PER-Event Masks (EM):** Bits 32-34 and 36 specify which types of events are recognized. The bits are assigned as follows:

- Bit 32: Successful-branching event
- Bit 33: Instruction-fetching event
- Bit 34: Storage-alteration event
- Bit 36: Store-using-real-address event (bit 34 must be one also)

Bits 32-34 and bit 36, when ones, specify that the corresponding types of events be recognized. However, bit 36 is effective for this purpose only when bit 34 is also one. When bit 34 is one, the storage-alteration event is recognized. When bits 34 and 36 are ones, both the storage-alteration event and the store-using-real-address event are recognized. When a bit is zero, the corresponding type of event is not recognized. When bit 34 is zero, both the storage-alteration event and the store-using-real-address event are not recognized.

**Branch-Address Control (B):** Bit 40 of control register 9 specifies, when one, that successful-branching events occur only for branches that are to a location within the designated storage area. When bit 40 is zero, successful branching events occur regardless of the branch-target address.

**Storage-Alteration-Space Control (S):** Bit 42 of control register 9 specifies, when one, that storage-alteration events occur as a result of references to the designated storage area only within designated address spaces. An address space is designated as one for which storage-alteration events occur by means of the storage-alteration-event bit in the address-space-control element that is used to translate references to the address space. Bit 42 is ignored when DAT is off. When DAT is off or bit 42 is zero, storage-alteration events are not restricted to occurring for only particular address spaces.

**PER Starting Address:** Bits 0-63 of control register 10 are the address of the beginning of the designated storage area.

**PER Ending Address:** Bits 0-63 of control register 11 are the address of the end of the designated storage area.

The address-space-control element has one of the following formats:

**Region-Table or Segment-Table Designation (R=0):**

```
  Region-Table or Segment-Table Origin
0  52 54 58 60 63
```

**Real-Space Designation (R=1):**

```
  Real-Space Token Origin
0  52 54 58 63
```

**Storage-Alteration-Event Bit (S):** When the storage-alteration-space control in control register 9 is one, bit 56 of the address-space control element specifies, when one, that the address
space defined by the address-space-control element is one for which storage-alteration events can occur. Bit 56 is examined when the address-space-control element is used to perform dynamic-address translation for a storage-operand store reference. The address-space-control element may be the PASCE, SASCE, or HASCE in control register 1, 7, or 13, respectively, or it may be obtained from an ASN-second-table entry during access-register translation. Instead of being obtained from an ASN-second-table entry in main storage, bit 56 may be obtained from an ASN-second-table entry in the ART-lookaside buffer (ALB). Bit 56 is ignored when the storage-alteration-space control is zero.

**Programming Notes:**

1. Models may operate at reduced performance while the CPU is enabled for PER events. In order to ensure that CPU performance is not degraded because of the operation of the PER facility, programs that do not use it should disable the CPU for PER events by setting either the PER mask in the PSW to zero or the PER-event masks in control register 9 to zero, or both. No degradation due to PER occurs when either of these fields is zero.

2. Some degradation may be experienced on some models every time control registers 9, 10, and 11 are loaded, even when the CPU is disabled for PER events (see the programming note under "Storage-Area Designation").

**Operation**

PER is under control of bit 1 of the PSW, the PER mask. When the PER mask and a particular PER-event mask bit are all ones, the CPU is enabled for the corresponding type of event; otherwise, it is disabled. However, the CPU is enabled for the store-using-real-address event only when the storage-alteration mask bit and the store-using-real-address mask bit are both one.

An interruption due to a PER event normally occurs after the execution of the instruction responsible for the event. The occurrence of the event does not affect the execution of the instruction, which may be completed, partially completed, terminated, suppressed, or nullified. However, recognition of a storage-alteration event causes no more than 4K bytes to be stored beginning with the byte that caused the event, and this may result in partial completion of an interruptible instruction.

When the CPU is disabled for a particular PER event at the time it occurs, either by the PER mask in the PSW or by the masks in control register 9, the event is not recognized.

A change to the PER mask in the PSW or to the PER control fields in control registers 9, 10, and 11 affects PER starting with the execution of the immediately following instruction.

A change to the storage-alteration-event bit in an address-space-control element in control register 1, 7, or 13 also affects PER starting with the execution of the immediately following instruction. A change to the storage-alteration-event bit in an address-space-control element that may be obtained, during access-register translation, from an ASN-second-table entry in either main storage or the ALB does not necessarily have an immediate, if any, effect on PER. However, PER is affected immediately after PURGE ALB is executed.

If a PER event occurs during the execution of an instruction which changes the CPU from being enabled to being disabled for that type of event, that PER event is recognized.

PER events may be recognized in a trial execution of an instruction, and subsequently the instruction, DAT-table entries, and operands may be refetched for the actual execution. If any refetched field was modified by another CPU or by a channel program between the trial execution and the actual execution, it is unpredictable whether the PER events indicated are for the trial or the actual execution.

For special-purpose instructions that are not described in this publication, the operation of PER may not be exactly as described in this section.

**Identification of Cause**

A program interruption for PER sets bit 8 of the interruption code to one and places identifying information in real storage locations 150-159 and in location 161 if the PER event is a storage-alteration event. Additional information is provided by means of the instruction address in the program old PSW and the ILC. The information...
stored in real locations 150-159 and 161 has the following format:

Locations 150-151:

<table>
<thead>
<tr>
<th>PERC</th>
<th>000</th>
<th>ATMID</th>
<th>AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>5</td>
<td>8</td>
<td>13</td>
</tr>
</tbody>
</table>

Locations 152-159:

<table>
<thead>
<tr>
<th>PER Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

Location 161:

<table>
<thead>
<tr>
<th>0000</th>
<th>PAID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

**PER Code (PERC):**  The occurrence of PER events is indicated by ones in bit positions 0-2 and 4. The bit position in the PER code for a particular type of event is 32 less than the bit position for that event in the PER-event-mask field in control register 9, except that a one in bit position 2 and a zero in bit position 4 of location 150 indicate a storage-alteration event, while ones in bit positions 2 and 4 indicate a store-using-real-address event. When a program interruption occurs, more than one type of PER event can be concurrently indicated. Additionally, if another program-interruption condition exists, the interruption code for the program interruption may indicate both the PER events and the other condition.

**Addressing-and-Translation-Mode Identification (ATMID):** During a program interruption when a PER event is indicated, bits 31, 32, 5, 16, and 17 of the PSW at the beginning of the execution of the instruction that caused the event may be stored in bit positions 8 and 10-13, respectively, of real locations 150-151. If bits 31, 32, 5, 16, and 17 are stored, then a one bit is stored in bit position 9 of locations 150-151. If bits 31, 32, 5, 16, and 17 are not stored, then zero bits are stored in bit positions 8-13 of locations 150-151.

Bits 8-13 of real locations 150-151 are named the addressing-and-translation-mode identification (ATMID). Bit 9 is named the ATMID-validity bit. When bit 9 is zero, it indicates that an invalid ATMID (all zeros) was stored.

The meanings of the bits of a valid ATMID are as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>PSW bit 31</td>
</tr>
<tr>
<td>9</td>
<td>ATMID-validity</td>
</tr>
<tr>
<td>10</td>
<td>PSW bit 32</td>
</tr>
<tr>
<td>11</td>
<td>PSW bit 5</td>
</tr>
<tr>
<td>12</td>
<td>PSW bit 16</td>
</tr>
<tr>
<td>13</td>
<td>PSW bit 17</td>
</tr>
</tbody>
</table>

A valid ATMID is necessarily stored only if the PER event was caused by one of the following instructions:

- BRANCH AND SAVE AND SET MODE (BASSM)
- BRANCH AND SET AUTHORITY (BSA)
- BRANCH AND SET MODE (BSM)
- BRANCH IN SUBSPACE GROUP (BSG)
- LOAD PSW (LPSW)
- LOAD PSW EXTENDED (LPSWE)
- PROGRAM CALL (PC)
- PROGRAM RETURN (PR)
- PROGRAM TRANSFER (PT)
- PROGRAM TRANSFER WITH INSTANCE (PTI)
- RESUME PROGRAM (RP)
- SET ADDRESS SPACE CONTROL (SAC)
- SET ADDRESS SPACE CONTROL FAST (SACF)
- SET ADDRESSING MODE (SAM24, SAM31, SAM64)
- SET SYSTEM MASK (SSM)
- STORE THEN AND SYSTEM MASK (STNSM)
- STORE THEN OR SYSTEM MASK (STOSM)
- SUPERVISOR CALL (SVC)
- TRAP (TRAP2, TRAP4)

It is unpredictable whether a valid ATMID is stored if the PER event was caused by any other instruction.

**PER ASCE Identification (AI):** If a storage-alteration event is indicated in the PER code (bit 2 is one and bit 4 is zero) and this event occurred when DAT was on, bits 14 and 15 of locations 150-151 are set to identify the address-space-control element (ASCE) that was used to translate the reference that caused the event, as follows:
Bits 14-15 Meaning
00 Primary ASCE was used.
01 An AR-specified ASCE was used. The PER access id, real location 161, can be examined to determine the ASCE used. However, if the primary, secondary, or home ASCE was used, bits 14 and 15 may be set to 00, 10, or 11, respectively, instead of to 01.
10 Secondary ASCE was used.
11 Home ASCE was used.

The CPU may avoid setting bits 14 and 15 to 01 by recognizing that access-list-entry token (ALET) 00000000 or 00000001 hex was used or that the ALET designated, through an access-list entry, an ASN-second-table entry containing an ASCE equal to the primary ASCE, secondary ASCE, or home ASCE.

If a storage-alteration event is not indicated in the PER code (bit 2 is zero or bit 4 is one) or DAT was off, zeros are stored in bit positions 14 and 15.

Zeros are stored in bit positions 3 and 5-7 of locations 150-151.

**PER Address:** The PER-address field at locations 152-159 contains the instruction address used to fetch the instruction in execution when one or more PER events were recognized. When the instruction is the target of EXECUTE, the instruction address used to fetch the EXECUTE instruction is placed in the PER-address field.

**PER Access Identification (PAID):** If a storage-alteration event is indicated in the PER code, an indication of the address space to which the event applies may be stored at location 161. If the access used an AR-specified address-space-control element, the number of the access register used is stored in bit positions 4-7 of location 161, and zeros are stored in bit positions 0-3. The contents of location 161 are unpredictable if (1) the CPU was in the access-register mode but the access was an implicit reference to the linkage stack or (2) the CPU was not in the access-register mode.

**Instruction Address:** The instruction address in the program old PSW is the address of the instruction which would have been executed next, unless another program condition is also indicated, in which case the instruction address is that determined by the instruction ending due to that condition.

**ILC:** The ILC indicates the length of the instruction designated by the PER address, except when a concurrent specification exception for the PSW introduced by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or a supervisor-call interruption sets an ILC of 0.

**Programming Notes:**

1. PSW bit 31 is the extended-addressing-mode bit, and PSW bit 32 is the basic-addressing-mode bit. When PSW bit 31 and 32 are both one, they specify the 64-bit addressing mode. When PSW bit 31 is zero, PSW 32 specifies the 24-bit addressing mode if the bit is zero or the 31-bit addressing mode if the bit is one. PSW bit 5 is the DAT-mode bit, and PSW bits 16 and 17 are the address-space-control bits. For the handling of instruction and logical addresses in the different translation modes, see "Translation Modes" on page 3-35.

2. A valid ATMID allows the program handling the PER event to determine the address space from which the instruction that caused the event was fetched and also to determine which translation mode applied to the storage-operand references of the instruction, if any. Each of the instructions for which a valid ATMID is necessarily stored can change one or more of PSW bits 5, 16, and 17, with the result that the values of those bits in the program old PSW that is stored because of the PER event are not necessarily the values that existed at the beginning of the execution of the instruction that caused the event. The instructions for which a valid ATMID is necessarily stored are the only instructions that can change any of PSW bits 5, 16, and 17.

3. If a storage-alteration PER event is indicated and DAT was on when the event occurred, an indication of the address-space-control element that was used to translate the reference that caused the event is given by the PER ASCE identification, bits 14 and 15 of real locations 150-151. If bits 14 and 15 indicate that an AR-specified address-space-control element was used, the PER access identification in real location 161 can...
be used to determine the address space that was referenced. To determine if DAT was on, the program handling the PER event should first examine the ATMID-validity bit to determine whether a valid ATMID was stored and, if it was stored, then examine the DAT-mode bit in the ATMID. If a valid ATMID was not stored, the program should examine the DAT-mode bit in the program old PSW.

4. If a valid ATMID is stored, it also allows the program handling the PER event to determine the addressing mode (24-bit, 31-bit, or 64-bit) that existed for the instruction that caused the PER event. This knowledge of the addressing mode allows the program to determine, without any chance of error, the meaning of one bits in bit positions 0-39 of the addresses of the instruction and of the storage operands, if any, of the instruction and, thus, to determine accurately the locations of the instruction and operands. Note that the address of the instruction is not necessarily provided without error by the PER address in real locations 152-159 because that address may be the address of an EXECUTE instruction, with the address of the target instruction still to be determined from the fields that specify the second-operand address of the EXECUTE instruction. Also note that another possible source of error is that, in the 24-bit or 31-bit addressing mode, an instruction or operand may wrap around in storage by beginning just below the 16M-byte or 2G-byte boundary, respectively.

5. A valid ATMID is necessarily stored for all instructions that can change the addressing-mode bits. However, the ATMID mechanism does not provide complete assurance that the instruction causing a PER event and the instruction's operands can be located accurately because LOAD CONTROL and LOAD ADDRESS SPACE PARAMETERS can change the address-space-control element that was used to fetch the instruction.

In the case of an instruction-fetching event for SUPERVISOR CALL, the program interruption occurs immediately after the supervisor-call interruption.

If a PER event is recognized during the execution of an instruction which also introduces a new PSW with the type of PSW-format error which is recognized early (see "Exceptions Associated with the PSW" on page 6-9), both the specification exception and PER are indicated concurrently in the interruption code of the program interruption. If the PSW-format error is of the type which is recognized late, only PER is indicated in the interruption code. In both cases, the invalid PSW is stored as the program old PSW.

Recognition of a PER event does not normally affect the ending of instruction execution. However, in the following cases, execution of an interruptible instruction is not completed normally:

1. When the instruction is due to be interrupted for an asynchronous condition (I/O, external, restart, or repressible machine-check condition), a program interruption for the PER event occurs first, and the other interruptions occur subsequently (subject to the mask bits in the new PSW) in the normal priority order.

2. When the stop function is performed, a program interruption indicating the PER event occurs before the CPU enters the stopped state.

3. When any program exception is recognized, PER events recognized for that instruction execution are indicated concurrently.

4. Depending on the model, in certain situations, recognition of a PER event may appear to cause the instruction to be interrupted prematurely without concurrent indication of a program exception, without an interruption for any asynchronous condition, and without the CPU entering the stopped state. In particular, recognition of a storage-alteration event causes no more than 4K bytes to be stored beginning with the byte that caused the event.

In cases 1 and 2 above, if the only PER event that has been recognized is an instruction-fetching event and another unit of operation of the instruction remains to be executed, the event may be discarded, with the result that a program inter-
rupation does not occur. Whether the event is dis-
carded is unpredictable.

Programming Notes:

1. In the following cases, an instruction can both
cause a program interruption for a PER event
and change the value of fields controlling an
interruption for PER events. The original field
values determine whether a program inter-
ruption takes place for the PER event.

a. The instructions LOAD PSW, LOAD PSW
EXTENDED, SET SYSTEM MASK,
STORE THEN AND SYSTEM MASK, and
SUPERVISOR CALL can cause an
instruction-fetching event and disable the
CPU for PER interruptions. Additionally,
STORE THEN AND SYSTEM MASK can
cause a storage-alteration event to be
indicated. In all these cases, the program
old PSW associated with the program
interruption for the PER event may indi-
cate that the CPU was disabled for PER
events.

b. An instruction-fetching event may be
recognized during execution of a LOAD
CONTROL instruction that changes the
value of the PER-event masks in control
register 9 or the addresses in control reg-
isters 10 and 11 controlling indication of
instruction-fetching events.

c. In the access-register mode, a storage-
alteration event that is permitted by a one
value of the storage-alteration-event bit in
an address-space-control element in an
ASN-second-table entry (designated by an
access-list entry) may be caused by any
store-type instruction that changes the
value of the bit from one to zero.

2. When a PER interruption occurs during
the execution of an interruptible instruction, the
ILC indicates the length of that instruction or
EXECUTE, as appropriate. When a PER
interruption occurs as a result of LOAD PSW,
LOAD PSW EXTENDED, PROGRAM
RETURN, or SUPERVISOR CALL, the ILC
indicates the length of these instructions or
EXECUTE, as appropriate, unless a concur-
rent specification exception on LOAD PSW,
LOAD PSW EXTENDED, or PROGRAM
RETURN calls for an ILC of 0.

3. When a PER interruption is caused by
branching, the PER address identifies the
branch instruction (or EXECUTE, as appro-
priate), whereas the old PSW points to the
next instruction to be executed. When the
interruption occurs during the execution of an
interruptible instruction, the PER address and
the instruction address in the old PSW are the
same.

Storage-Area Designation

Two types of PER events — instruction fetching
and storage alteration — always involve the des-
ignation of an area in storage. Successful-
branching events may involve this designation.
The storage area starts at the location designated
by the starting address in control register 10 and
extends up to and including the location desig-
nated by the ending address in control register 11.
The area extends to the right of the starting
address.

An instruction-fetching event occurs whenever the
first byte of an instruction or the first byte of the
target of an EXECUTE instruction is fetched from
the designated area. A storage-alteration event
occurs when a store access is made to the desig-
nated area by using an operand address that is
defined to be a logical or a virtual address.
However, when DAT is on and the storage-
alteration-space control in control register 9 is one,
a storage-alteration event occurs only when the
storage area is within an address space for which
the storage-alteration-event bit in the address-
space-control element is one. A storage-alteration
event does not occur for a store access made with
an operand address defined to be a real address.
When the branch-address control in control reg-
ister 9 is one, a successful-branching event occurs
when the first byte of the branch-target instruction
is within the designated area.

The set of addresses designated for successful-
branching, instruction-fetching, and storage-
alteration events wraps around at address
$2^{64} - 1$; that is, address 0 is considered to follow
address $2^{64} - 1$. When the starting address is
less than the ending address, the area is contig-
uous. When the starting address is greater than
the ending address, the set of locations design-
nated includes the area from the starting address
to address $2^{64} - 1$ and the area from address 0
to, and including, the ending address. When the starting address is equal to the ending address, only that one location is designated.

Address comparison for successful-branching, instruction-fetching, and storage-alteration events is always performed using 64-bit addresses. This is accomplished in the 24-bit or 31-bit addressing mode by extending the virtual, logical, or instruction address on the left with 39 or 33 zeros, respectively, before comparing it with the starting and ending addresses.

**Programming Note:** In some models, performance of address-range checking is assisted by means of an extension to each page-table entry in the TLB. In such an implementation, changing the contents of control registers 10 and 11 when the successful-branching, instruction-fetching, or storage-alteration-event mask is one, or setting any of these PER-event masks to one, may cause the TLB to be cleared of entries. This degradation may be experienced even when the CPU is disabled for PER events. Thus, when possible, the program should avoid loading control registers 9, 10, or 11.

**PER Events**

### Successful Branching

When the branch-address control in control register 9 is zero, a successful-branching event occurs independent of the branch-target address. When the branch-address control is one, a successful-branching event occurs only when the first byte of the branch-target instruction is in the storage area designated by control registers 10 and 11.

Subject to the effect of the branch-address control, a successful-branching event occurs whenever one of the following instructions causes branching:

- BRANCH AND LINK (BAL, BALR)
- BRANCH AND SAVE (BAS, BASR)
- BRANCH AND SAVE AND SET MODE (BASSM)
- BRANCH AND SET AUTHORITY (BSA)
- BRANCH AND SET MODE (BSM)
- BRANCH AND STACK (BAK)
- BRANCH IN SUBSPACE GROUP (BSG)
- BRANCH ON CONDITION (BC, BCR)
- BRANCH ON COUNT (BCT, BCTR, BCTG, BCTGR)
- BRANCH ON INDEX HIGH (BXH, BXHG)
- BRANCH ON INDEX LOW OR EQUAL (BXLE, BXLEG)
- BRANCH RELATIVE AND SAVE (BRAS)
- BRANCH RELATIVE AND SAVE LONG (BRASL)
- BRANCH RELATIVE ON CONDITION (BRC)
- BRANCH RELATIVE ON CONDITION LONG (BRCL)
- BRANCH RELATIVE ON COUNT (BRCT, BRCTG)
- BRANCH RELATIVE ON INDEX HIGH (BRXH, BRXHG)
- BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLE, BRXLG)
- RESUME PROGRAM (RP)
- TRAP (TRAP2, TRAP4)

Subject to the effect of the branch-address control, a successful-branching event also occurs whenever one of the following instructions causes branching:

- PROGRAM CALL (PC)
- PROGRAM RETURN (PR)
- PROGRAM TRANSFER (PT)
- PROGRAM TRANSFER WITH INSTANCE (PTI)

For PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, the branch-target address is considered to be the new instruction address that is placed in the PSW by the instruction.

A successful-branching event causes a PER successful-branching event to be recognized if bit 32 of the PER-event masks is one and the PER mask in the PSW is one.

A PER successful-branching event is indicated by setting bit 0 of the PER code to one.

### Instruction Fetching

An instruction-fetching event occurs if the first byte of the instruction is within the storage area designated by control registers 10 and 11. An instruction-fetching event also occurs if the first byte of the target of EXECUTE is within the designated storage area.

An instruction-fetching event causes a PER instruction-fetching event to be recognized if bit 33
of the PER-event masks is one and the PER mask in the PSW is one.

If an instruction-fetching event is the only PER event recognized for an interruptible instruction that is to be interrupted because of an asynchronous condition (I/O, external, restart, or repressible machine-check condition) or the performance of the stop function, and if a unit of operation of the instruction remains to be executed, the instruction-fetching event may be discarded, and whether it is discarded is unpredictable.

The PER instruction-fetching event is indicated by setting bit 1 of the PER code to one.

Storage Alteration

A storage-alteration event occurs whenever a CPU, by using a logical or virtual address, makes a store access without an access exception to the storage area designated by control registers 10 and 11. However, when DAT is on and the storage-alteration-space control in control register 9 is one, the event occurs only if the storage-alteration-event bit is one in the address-space-control element that is used by DAT to translate the reference to the storage location.

The contents of storage are considered to have been altered whenever the CPU executes an instruction that causes all or part of an operand to be stored within the designated storage area. Alteration is considered to take place whenever storing is considered to take place for purposes of indicating protection exceptions, except that recognition does not occur for the storing of data by a channel program. (See "Recognition of Access Exceptions" on page 6-36) Storing constitutes alteration for PER purposes even if the value stored is the same as the original value.

Implied locations that are referred to by the CPU in the process of performing an interruption are not monitored. Such locations include PSW and interruption-code locations. These locations, however, are monitored when information is stored there explicitly by an instruction. Similarly, monitoring does not apply to the storing of data by a channel program. Implied locations in the linkage stack, which are stored in by instructions that operate on the linkage stack, are monitored.

The I/O instructions are considered to alter the second-operand location only when storing actually occurs.

Storage alteration does not apply to instructions whose operands are specified to have real addresses. Thus, storage alteration does not apply to INVALIDATE PAGE TABLE ENTRY, RESET REFERENCE BIT EXTENDED, SET STORAGE KEY EXTENDED, STORE USING REAL ADDRESS, TEST BLOCK, and TEST PENDING INTERRUPTION (when the effective address is zero).

A storage-alteration event causes a PER storage-alteration event to be recognized if bit 34 of the PER-event masks is one and the PER mask in the PSW is one. Bit 36 of the PER-event masks is ignored when determining whether a PER storage-alteration event is to be recognized.

A PER storage-alteration event is indicated by setting bit 2 of the PER code to one and bit 4 of the PER code to zero.

Store Using Real Address

A store-using-real-address event occurs whenever the STORE USING REAL ADDRESS instruction is executed.

There is no relationship between the store-using-real-address event and the designated storage area.

A store-using-real-address event causes a PER store-using-real-address event to be recognized if bits 34 and 36 of the PER-event mask are ones and the PER mask in the PSW is one.

A PER store-using-real-address event is indicated by setting bits 2 and 4 of the PER code to one.

Indication of PER Events Concurrently with Other Interruption Conditions

The following rules govern the indication of PER events caused by an instruction that also causes a program exception, a monitor event, a space-switch event, or a supervisor-call interruption.

1. The indication of an instruction-fetching event does not depend on whether the execution of the instruction was completed, terminated,
suppressed, or nullified. However, special cases of suppression and nullification are as follows:

a. When the instruction is designated by an odd instruction address in the PSW, the instruction-fetching event is not indicated.

b. When an access exception applies to the first, second, or third halfword of the instruction, it is unpredictable whether the instruction-fetching event is indicated.

c. When the target address of EXECUTE is odd or an access exception applies to the first, second, or third halfword of the target instruction, it is unpredictable whether the instruction-fetching event is indicated for the target instruction, and it is also unpredictable whether the event is indicated for the EXECUTE instruction.

2. When the operation is completed or partially completed, the event is indicated, regardless of whether any program exception, space-switch event, or monitor event is also recognized.

3. Successful branching, storage alteration, and store using real address are not indicated for an operation or, in case the instruction is interruptible, for a unit of operation that is suppressed or nullified.

4. When the execution of the instruction is terminated, storage alteration is indicated whenever the event has occurred, and a model may indicate the event if the event would have occurred had the execution of the instruction been completed, even if altering the contents of the result field is contingent on operand values. For purposes of this definition, the occurrence of those exceptions which permit termination (addressing, protection, and data) is considered to cause termination, even if no result area is changed.

5. When LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, SET SYSTEM MASK, STORE THEN OR SYSTEM MASK, or SUPERVISOR CALL causes a PER condition and at the same time introduces a new PSW with the type of PSW-format error that is recognized as part of the execution of the following instruction, the introduced PSW is stored as the old PSW without the following instruction being fetched or executed and without the specification exception being recognized.

The indication of PER events concurrently with other program-interruption conditions for the same instruction, as described in cases 1-4 above, is summarized in Figure 4-8 on page 4-38. Cases 5 and 6 are shown in Figure 4-9 on page 4-39.

Programming Notes:

1. The execution of the interruptible instructions MOVE LONG, TEST BLOCK, and COMPARE LOGICAL LONG can cause events for instruction fetching. Additionally, MOVE LONG can cause the storage-alteration event.

Interruption of such an instruction may cause a PER event to be indicated more than once. It may be necessary, therefore, for a program to remove the redundant event indications from the PER data. The following rules govern the indication of the applicable events during execution of these instructions:

a. The instruction-fetching event is indicated whenever the instruction is fetched for execution, regardless of whether it is the initial execution or a resumption, except that the event may be discarded (not indicated) if it is the only PER event to be indicated, the interruption is due to an asynchronous interruption condition or the performance of the stop function, and a unit of operation of the instruction remains to be executed.

b. The storage-alteration event is indicated only when data has been stored in the designated storage area by the portion of the operation starting with the last initiation and ending with the last byte transferred before the interruption. No special indication is provided on premature interruptions as to whether the event will occur again upon the resumption of the operation. When the designated storage area is a single byte location, a storage-alteration
event can be recognized only once in the execution of MOVE LONG.

2. The following is an outline of the general action a program must take to delete multiple entries in the PER data for an interruptible instruction so that only one entry for each complete execution of the instruction is obtained:
   a. Check to see if the PER address is equal to the instruction address in the old PSW and if the last instruction executed was interruptible.
   b. If both conditions are met, delete instruction-fetching events.
   c. If both conditions are met and the event is storage alteration, delete the event if some part of the remaining destination operand is within the designated storage area.

3. An example of the indication of an instruction-fetching PER event caused by either a LOAD PSW (or LOAD PSW EXTENDED) instruction or the following instruction, in connection with an early PSW-format error or odd instruction address introduced by the LOAD PSW instruction, is shown in Figure 4-9 on page 4-39.
<table>
<thead>
<tr>
<th>Concurrent Condition</th>
<th>Type of Ending</th>
<th>PER Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Odd instruction address in the PSW</td>
<td>S</td>
<td>No</td>
</tr>
<tr>
<td>Instruction access</td>
<td>N or S</td>
<td>U</td>
</tr>
<tr>
<td>EXECUTE target address odd</td>
<td>S</td>
<td>No</td>
</tr>
<tr>
<td>EXECUTE target access</td>
<td>N or S</td>
<td>U</td>
</tr>
<tr>
<td>Other nullifying</td>
<td>N</td>
<td>Yes</td>
</tr>
<tr>
<td>Other suppressing</td>
<td>S</td>
<td>Yes</td>
</tr>
<tr>
<td>All terminating</td>
<td>T</td>
<td>Yes</td>
</tr>
<tr>
<td>All completing</td>
<td>C</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Explanation:**

- The condition does not apply.

1 Although PER events of this type are not indicated for the current unit of operation of an interruptible instruction, PER events of this type that were recognized on completed units of operation of the interruptible instruction are indicated.

2 This event may be indicated, depending on the model, if the event has not occurred but would have been indicated if execution had been completed.

- The operation, or, in the case of the interruptible instructions, the unit of operation is completed.

- The operation, or, in the case of the interruptible instructions, the unit of operation is nullified.

- The operation, or, in the case of the interruptible instructions, the unit of operation is suppressed.

- The execution of the instruction is terminated.

Yes The PER event is indicated with the other program interruption condition if the event has occurred; that is, the instruction address in the PSW was replaced and the branch-address control and designated storage area allow the event occurrence, an attempt was made to execute an instruction whose first byte is located in the designated storage area, or the contents of the designated storage area were altered.

No The PER event is not indicated.

U It is unpredictable whether the PER event is indicated.

*Figure 4-8. Indication of PER Events with Other Concurrent Conditions*
Figure 4-9. Example of Instruction-Fetching PER Event and Early PSW-Format Error or Odd Instruction Address
Timing
The timing facilities include three facilities for measuring time: the TOD clock, the clock comparator, and the CPU timer. A TOD programmable register is associated with the TOD clock.

In a multiprocessing configuration, a single TOD clock is shared by all CPUs. Each CPU has its own clock comparator, CPU timer, and TOD programmable register.

Time-of-Day Clock
The time-of-day (TOD) clock provides a high-resolution measure of real time suitable for the indication of date and time of day. The cycle of the clock is approximately 143 years. A single TOD clock is shared by all CPUs in the configuration.

Format
The TOD clock is a 104-bit register. It is a binary counter with the format shown in the following illustration.

```
0 51 64 103
```

The TOD clock nominally is incremented by adding a one in bit position 51 every microsecond. In models having a higher or lower resolution, a different bit position is incremented at such a frequency that the rate of advancing the clock is the same as if a one were added in bit position 51 every microsecond. The resolution of the TOD clock is such that the incrementing rate is comparable to the instruction-execution rate of the model.

When incrementing of the clock causes a carry to be propagated out of bit position 0, the carry is ignored, and counting continues from zero. The program is not alerted, and no interruption condition is generated as a result of the overflow.

The operation of the clock is not affected by any normal activity or event in the system. Incrementing of the clock does not depend on whether the wait-state bit of the PSW is one or whether the CPU is in the operating, load, stopped, or check-stop state. Its operation is not affected by CPU, initial-CPU, or clear resets or by initial program loading. Operation of the clock is also not affected by the setting of the rate control or by an initial-machine-loading operation. Depending on the model and the configuration, the TOD clock may or may not be powered independent of the CPU.

States
The following states are distinguished for the TOD clock: set, not set, stopped, error, and not operational. The state determines the condition code set by execution of STORE CLOCK and STORE CLOCK EXTENDED. The clock is incremented, and is said to be running, when it is in either the set state or the not-set state.

Not-Set State: When the power for the clock is turned on, the clock is set to zero, and the clock enters the not-set state. The clock is incremented when in the not-set state.

When the clock is in the not-set state, execution of STORE CLOCK or STORE CLOCK EXTENDED causes condition code 1 to be set and the current value of the running clock to be stored.

Stopped State: The clock enters the stopped state when SET CLOCK is executed and the execution results in the clock being set. This occurs when SET CLOCK is executed without encountering any exceptions and either any manual TOD-clock control in the configuration is set to the enable-set position or the TOD-clock-control-override control, bit 42 of control register 14, is one. The clock can be placed in the stopped state from the set, not-set, and error states. The clock is not incremented while in the stopped state.

When the clock is in the stopped state, execution of STORE CLOCK or STORE CLOCK EXTENDED causes condition code 3 to be set and the value of the stopped clock to be stored.

Set State: The clock enters the set state only from the stopped state. The change of state is under control of the TOD-clock-sync-control bit, bit 34 of control register 0, of the CPU which most recently caused the clock to enter the stopped state. If the bit is zero, the clock enters the set state at the completion of execution of SET CLOCK. If the bit is one, the clock remains in the
stopped state until the bit is set to zero on that CPU or until another CPU executes a SET CLOCK instruction affecting the clock. If an external time reference (ETR) is installed, a signal from the ETR may be used to set the set state from the stopped state.

Incrementing of the clock begins with the first stepping pulse after the clock enters the set state.

When the clock is in the set state, execution of STORE CLOCK or STORE CLOCK EXTENDED causes condition code 0 to be set and the current value of the running clock to be stored.

**Error State:** The clock enters the error state when a malfunction is detected that is likely to have affected the validity of the clock value. It depends on the model whether the clock can be placed in this state. A timing-facility-damage machine-check-interruption condition is generated on each CPU in the configuration whenever the clock enters the error state.

When STORE CLOCK or STORE CLOCK EXTENDED is executed and the clock is in the error state, condition code 2 is set, and the value stored is zero.

**Not-Operational State:** The clock is in the not-operational state when its power is off or when it is disabled for maintenance. It depends on the model whether the clock can be placed in this state. Whenever the clock enters the not-operational state, a timing-facility-damage machine-check-interruption condition is generated on each CPU in the configuration.

When the clock is in the not-operational state, execution of STORE CLOCK or STORE CLOCK EXTENDED causes condition code 3 to be set, and zero is stored.

**Changes in Clock State**

When the TOD clock changes value because of the execution of SET CLOCK or changes state, interruption conditions pending for the clock comparator and CPU timer may or may not be recognized for up to 1.048576 seconds ($2^{20}$ microseconds) after the change.

The results of channel-subsystem-monitoring-facility operations may be unpredictable as a result of changes to the TOD clock.

**Setting and Inspecting the Clock**

The clock can be set to a specified value by execution of SET CLOCK if the manual TOD-clock control of any CPU in the configuration is in the enable-set position or the TOD-clock-control-override control, bit 42 of control register 14, is one. SET CLOCK sets bits of the clock with the contents of corresponding bit positions of a doubleword operand in storage.

Setting the clock replaces the values in all bit positions from bit position 0 through the rightmost position that is incremented when the clock is running. However, on some models, the rightmost bits starting at or to the right of bit 52 of the specified value are ignored, and zeros are placed in the corresponding positions of the clock. Zeros are also placed in positions to the right of bit position 63 of the clock.

The TOD clock can be inspected by executing STORE CLOCK, which causes bits 0-63 of the clock to be stored in an eight-byte operand in storage, or by executing STORE CLOCK EXTENDED, which causes bits 0-103 of the clock to be stored in bytes 1-13 of a 16-byte operand in storage. STORE CLOCK EXTENDED stores zeros in the leftmost byte, byte 0, of its storage operand, and it obtains the TOD programmable field from bit positions 16-31 of the TOD programmable register and stores it in byte positions 14 and 15 of the storage operand. The operand stored by STORE CLOCK EXTENDED has the following format:

<table>
<thead>
<tr>
<th>Zeros</th>
<th>TOD Clock</th>
<th>Programmable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 8 112 127</td>
</tr>
</tbody>
</table>

At some time in the future, STORE CLOCK EXTENDED on new models will store a leftmost extension of the TOD clock in byte position 0 of its storage operand; see programming note 14 on page 4-44.

Two executions of STORE CLOCK or STORE CLOCK EXTENDED, possibly on different CPUs in the same configuration, always store different values of the clock if the clock is running. If the clock is stopped, zeros are stored in the clock value, bits 8-111 of the storage operand, in posi-
tions to the right of the rightmost bit position that is incremented when the clock is running. The programmable field continues to be stored even when the clock is stopped.

The values stored for a running clock by STORE CLOCK or STORE CLOCK EXTENDED always correctly imply the sequence of execution of these instructions by one or more CPUs for all cases where the sequence can be discovered by the program. To ensure that unique values are obtained when the value of a running clock is stored, nonzero values may be stored in positions to the right of the rightmost incremented bit position. When the value of a running clock is stored by STORE CLOCK EXTENDED, the value in bit positions 64-103 of the clock (bit positions 72-111 of the storage operand) is always nonzero; this ensures that values stored by STORE CLOCK EXTENDED are always unique when compared with values stored by STORE CLOCK and extended on the right with zeros.

For the purpose of establishing uniqueness and sequence of occurrence of the results of STORE CLOCK and STORE CLOCK EXTENDED, the 64-bit value provided by STORE CLOCK may be considered to be extended to 104 bits by appending 40 zeros on the right, with the STORE CLOCK value and STORE CLOCK EXTENDED bits 8-111 then both being treated as 104-bit unsigned binary integers.

In a configuration where more than one CPU accesses the clock, SET CLOCK is interlocked such that the entire contents appear to be updated concurrently; that is, if SET CLOCK instructions are executed simultaneously by two CPUs, the final result is either one or the other value. If SET CLOCK is executed by one CPU and STORE CLOCK or STORE CLOCK EXTENDED by the other, the result obtained by STORE CLOCK or STORE CLOCK EXTENDED is either the entire old value or the entire new value. When SET CLOCK is executed by one CPU, a STORE CLOCK or STORE CLOCK EXTENDED instruction executed by another CPU may find the clock in the stopped state even when the TOD-clock-sync-control bit, bit 34 of control register 0, of each CPU is zero. Since the clock enters the set state before incrementing, the first STORE CLOCK or STORE CLOCK EXTENDED instruction executed after the clock enters the set state may still find the original value introduced by SET CLOCK.

TOD Programmable Register
Each CPU has a TOD programmable register. Bits 16-31 of the register contain the programmable field that is appended on the right to the TOD-clock value by STORE CLOCK EXTENDED. The register has the following format:

<table>
<thead>
<tr>
<th>Programmable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

The register is loaded by SET CLOCK PROGRAMMABLE FIELD. The contents of the register are reset to a value of all zeros by initial CPU reset.

Programming Notes:

1. Bit position 31 of the clock is incremented every 1.048576 seconds; for some applications, reference to the leftmost 32 bits of the clock may provide sufficient resolution.

2. Communication between systems is facilitated by establishing a standard time origin that is the calendar date and time to which a clock value of zero corresponds. January 1, 1900, 0 a.m. Coordinated Universal Time (UTC) is recommended as this origin, and it is said to begin the standard epoch for the clock. This is also the epoch used when the TOD clock is synchronized to the external time reference (ETR), and, for this reason, the epoch is sometimes referred to as ETR time. The former term, Greenwich Mean Time (GMT), is now obsolete and has been replaced with the more precise UTC.

3. Historically, one of the most important uses of standard time has been for navigation. Prior to 1972, standard time, then called GMT, was defined to have a variable-length second and was synchronized to within 100 milliseconds of the rotational position of the earth. Synchronization was accomplished by occasional changes in the length of the second, typically in parts per billion, and also by occasional insertion and deletion of small increments of time, typically 50 or 100 milliseconds. Beginning in 1972, a new standard time scale, called UTC, was defined to have a fixed-length second and be kept synchronized to
within 900 milliseconds of the rotational position of the earth by means of occasional adjustments of exactly one second called a leap second. The change from GMT to UTC occurred between the last second of the day on December 31, 1971 and the first second of the day on January 1, 1972 and included insertion of 107.758 milliseconds in the standard time scale to make UTC exactly 10 seconds behind International Atomic Time (TAI). For reasons of simplicity in this document, the term UTC is sometimes extrapolated backward before 1972 by assuming no time adjustments in that time scale before 1972. For the same reasons, conversion between ETR time and UTC does not take into consideration the time adjustments prior to 1972, and, thus, ETR time differs from TAI by a fixed amount of 10 seconds. Because of the occurrence of 22 leap seconds, UTC now is behind TAI by 32 seconds.

6. Because of the limited accuracy of manually setting the clock value, the rightmost bit positions of the clock, expressing fractions of a second, are normally not valid as indications of the time of day. However, they permit elapsed-time measurements of high resolution.

7. The following chart shows the time interval between instants at which various bit positions of the TOD clock are stepped. This time value may also be considered as the weighted time value that the bit, when one, represents.

<table>
<thead>
<tr>
<th>TOD-Clock Bit</th>
<th>Stepping Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Days</td>
<td>Hours</td>
</tr>
<tr>
<td>51</td>
<td>0.000 001</td>
</tr>
<tr>
<td>47</td>
<td>0.000 016</td>
</tr>
<tr>
<td>43</td>
<td>0.000 256</td>
</tr>
<tr>
<td>39</td>
<td>0.004 096</td>
</tr>
<tr>
<td>35</td>
<td>0.005 536</td>
</tr>
<tr>
<td>31</td>
<td>1.048 576</td>
</tr>
<tr>
<td>27</td>
<td>16.777 216</td>
</tr>
<tr>
<td>23</td>
<td>4 28.435 456</td>
</tr>
<tr>
<td>19</td>
<td>1 11 34.967 296</td>
</tr>
<tr>
<td>15</td>
<td>19 5 19.476 736</td>
</tr>
<tr>
<td>11</td>
<td>12 17 25 11.627 776</td>
</tr>
<tr>
<td>7</td>
<td>203 14 43 6.044 416</td>
</tr>
<tr>
<td>3</td>
<td>3257 19 29 36.710 656</td>
</tr>
</tbody>
</table>

8. The following chart shows the TOD clock setting for 00:00:00 (0 am), UTC time, for several dates: January 1, 1900, January 1, 1972, and for that instant in time just after each of the 22 leap seconds that have occurred through November, 2000. Each of these leap seconds was inserted in the UTC time scale beginning at 23:59:60 UTC of the day previous to the one listed and ending at 00:00:00 UTC of the day listed.
9. The stepping value of TOD-clock bit position 63, if implemented, is 2\textsuperscript{12} microseconds, or approximately 244 picoseconds. This value is called a clock unit.

The following chart shows various time intervals in clock units expressed in hexadecimal notation.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Clock Units (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 microsecond</td>
<td>1000</td>
</tr>
<tr>
<td>1 millisecond</td>
<td>3E 8000</td>
</tr>
<tr>
<td>1 second</td>
<td>F424 0000</td>
</tr>
<tr>
<td>1 minute</td>
<td>39 3870 0000</td>
</tr>
<tr>
<td>1 hour</td>
<td>D69 3A40 0000</td>
</tr>
<tr>
<td>1 day</td>
<td>1 41DD 7600 0000</td>
</tr>
<tr>
<td>365 days</td>
<td>1CA EBC1 3E00 0000</td>
</tr>
<tr>
<td>366 days</td>
<td>1CC 2A9E B400 0000</td>
</tr>
<tr>
<td>1,461 days*</td>
<td>72C E4E2 6E00 0000</td>
</tr>
</tbody>
</table>

* Number of days in four years, including a leap year. Note that the year 1900 was not a leap year. Thus, the four-year span starting in 1900 has only 1,460 days.

10. The charts in notes 6-8 are useful when examining the value stored by STORE CLOCK. Similar charts for use when examining the value stored by STORE CLOCK EXTENDED are in programming notes at the end of the definition of that instruction.

11. In a multiprocessing configuration, after the TOD clock is set and begins running, the program should delay activity for 2\textsuperscript{28} microseconds (1.048576 seconds) to ensure that the CPU-timer and clock-comparator interruption conditions are recognized by the CPU.

12. Due to the sequencing rules for the results of STORE CLOCK and STORE CLOCK EXTENDED, the execution of STORE CLOCK may be considerably slower than that of STORE CLOCK EXTENDED on models that incremented a bit position of the TOD clock to the right of position 63.

13. Uniqueness of TOD-clock values can be extended to apply to processors in separate configurations by including a configuration identification in the TOD programmable field.

14. At some time in the future, new models will use a carry from bit position 0 of the TOD clock to increment an additional eight-bit binary counter. STORE CLOCK EXTENDED will store the contents of this counter in byte position 0 of its storage operand. A variation of SET CLOCK will set the counter, as well as the TOD clock. Variations of SET CLOCK and STORE CLOCK EXTENDED, the execution of STORE CLOCK EXTENDED are in programming notes at the end of the definition of that instruction.

**TOD-Clock Synchronization**

The following functions are provided if an external time reference (ETR) is installed:

- A clock in the stopped state, with the TOD-clock-sync-control bit (bit 34 of control register 0) set to one, is placed in the set state and starts incrementing when an ETR signal occurs.
The stepping rates for the TOD clock and the ETR are synchronized.

Bits 32 through the rightmost incremented bit of a clock in the set state are compared with the same bits of the ETR. An unequal condition is signaled by an external-damage machine-check-interruption condition. The machine-check-interruption condition may not be recognized for up to \(2^{20}\) microseconds after the unequal condition occurs.

**Programming Notes:**

1. TOD-clock synchronization provides for synchronizing and checking only bits 32 through the rightmost incremented bit of the TOD clock. Bits 0-31 of the TOD clock may be different from those of the ETR.

2. If an ETR is installed, SET CLOCK must place all zeros in bit positions 32 through the rightmost incremented bit position of the TOD clock; otherwise, an external-damage machine-check-interruption condition will be recognized.

**Clock Comparator**

The clock comparator provides a means of causing an interruption when the TOD-clock value exceeds a value specified by the program.

In a configuration with more than one CPU, each CPU has a separate clock comparator.

The clock comparator has the same format as bits 0-63 of the TOD clock. The clock comparator nominally consists of bits 0-47, which are compared with the corresponding bits of the TOD clock. In some models, higher resolution is obtained by providing more than 48 bits. The bits in positions provided in the clock comparator are compared with the corresponding bits of the clock. When the resolution of the clock is less than that of the clock comparator, the contents of the clock comparator are compared with the clock value as this value would be stored by executing STORE CLOCK.

The clock comparator causes an external interruption with the interruption code 1004 hex. A request for a clock-comparator interruption exists whenever either of the following conditions exists:

1. The TOD clock is running and the value of the clock comparator is less than the value in the compared portion of the clock, both values being considered unsigned binary integers. Comparison follows the rules of unsigned binary arithmetic.

2. The TOD clock is in the error state or the not-operational state.

A request for a clock-comparator interruption does not remain pending when the value of the clock comparator is made equal to or greater than that of the TOD clock or when the value of the TOD clock is made less than the clock-comparator value. The latter may occur as a result of the TOD clock either being set or wrapping to zero.

The clock comparator can be inspected by executing the instruction STORE CLOCK COMPARATOR and can be set to a specified value by executing the SET CLOCK COMPARATOR instruction.

The contents of the clock comparator are initialized to zero by initial CPU reset.

**Programming Notes:**

1. An interruption request for the clock comparator persists as long as the clock-comparator value is less than that of the TOD clock or as long as the TOD clock is in the error state or the not-operational state. Therefore, one of the following actions must be taken after an external interruption for the clock comparator has occurred and before the CPU is again enabled for external interruptions: the value of the clock comparator must be replaced, the TOD clock must be set, the TOD clock must wrap to zero, or the clock-comparator-subclass mask must be set to zero. Otherwise, loops of external interruptions are formed.

2. The instruction STORE CLOCK or STORE CLOCK EXTENDED may store a value which is greater than that in the clock comparator, even though the CPU is enabled for the clock-comparator interruption. This is because the TOD clock may be incremented one or more times between when instruction execution is begun and when the clock value is accessed. In this situation, the interruption occurs when the execution of STORE CLOCK or STORE CLOCK EXTENDED is completed.
CPU Timer

The CPU timer provides a means for measuring elapsed CPU time and for causing an interruption when a specified amount of time has elapsed.

In a configuration with more than one CPU, each CPU has a separate CPU timer.

The CPU timer is a binary counter with a format which is the same as that of bits 0-63 of the TOD clock, except that bit 0 is considered a sign. The CPU timer nominally is decremented by subtracting a one in bit position 51 every microsecond. In models having a higher or lower resolution, a different bit position is decremented at such a frequency that the rate of decrementing the CPU timer is the same as if a one were subtracted in bit position 51 every microsecond. The resolution of the CPU timer is such that the stepping rate is comparable to the instruction-execution rate of the model.

The CPU timer requests an external interruption with the interruption code 1005 hex whenever the CPU-timer value is negative (bit 0 of the CPU timer is one). The request does not remain pending when the CPU-timer value is changed to a nonnegative value.

When both the CPU timer and the TOD clock are running, the stepping rates are synchronized such that both are stepped at the same rate. Normally, decrementing the CPU timer is not affected by concurrent I/O activity. However, in some models the CPU timer may stop during extreme I/O activity and other similar interference situations. In these cases, the time recorded by the CPU timer provides a more accurate measure of the CPU time used by the program than would have been recorded had the CPU timer continued to step.

The CPU timer is decremented when the CPU is in the operating state or the load state. When the manual rate control is set to instruction step, the CPU timer is decremented only during the time in which the CPU is actually performing a unit of operation. However, depending on the model, the CPU timer may or may not be decremented when the TOD clock is in the error, stopped, or non-operational state.

Depending on the model, the CPU timer may or may not be decremented when the CPU is in the check-stop state.

The CPU timer can be inspected by executing the instruction STORE CPU TIMER and can be set to a specified value by executing the SET CPU TIMER instruction.

The CPU timer is set to zero by initial CPU reset.

Programming Notes:

1. The CPU timer in association with a program may be used both to measure CPU-execution time and to signal the end of a time interval on the CPU.

2. The time measured for the execution of a sequence of instructions may depend on the effects of such things as I/O interference, the availability of pages, and instruction retry. Therefore, repeated measurements of the same sequence on the same installation may differ.

3. The fact that a CPU-timer interruption does not remain pending when the CPU timer is set to a positive value eliminates the problem of an undesired interruption. This would occur if, between the time when the old value is stored and a new value is set, the CPU is disabled for CPU-timer interruptions and the CPU timer value goes from positive to negative.

4. The fact that CPU-timer interruptions are requested whenever the CPU timer is negative (rather than just when the CPU timer goes from positive to negative) eliminates the requirement for testing a value to ensure that it is positive before setting the CPU timer to that value.

As an example, assume that a program being timed by the CPU timer is interrupted for a cause other than the CPU timer, external interruptions are disallowed by the new PSW, and the CPU-timer value is then saved by STORE CPU TIMER. This value could be negative if the CPU timer went from positive to negative since the interruption. Subsequently, when the program being timed is to continue, the CPU timer may be set to the saved value by SET CPU TIMER. A CPU-timer interruption occurs immediately after external interruptions are again enabled if the saved value was negative.
The persistence of the CPU-timer-interruption request means, however, that after an external interruption for the CPU timer has occurred, the value of the CPU timer must be replaced, the value in the CPU timer must wrap to a positive value, or the CPU-timer-subclass mask must be set to zero before the CPU is again enabled for external interruptions. Otherwise, loops of external interruptions are formed.

5. The instruction STORE CPU TIMER may store a negative value even though the CPU is enabled for the interruption. This is because the CPU-timer value may be decremented one or more times between when instruction execution is begun and when the CPU timer is accessed. In this situation, the interruption occurs when the execution of STORE CPU TIMER is completed.

**Externally Initiated Functions**

**Resets**

Five reset functions are provided:

- CPU reset
- Initial CPU reset
- Subsystem reset
- Clear reset
- Power-on reset

CPU reset provides a means of clearing equipment-check indications and any resultant unpredictability in the CPU state with the least amount of information destroyed. In particular, it is used to clear check conditions when the CPU state is to be preserved for analysis or resumption of the operation. CPU reset sets the architectural mode to the ESA/390 mode if it is caused by activation of the load-normal key. When CPU reset sets the ESA/390 mode, it saves the current PSW so that PSW can be restored by a SIGNAL PROCESSOR set-architecture order that changes the architectural mode back to z/Architecture.

Initial CPU reset provides the functions of CPU reset together with initialization of the current PSW, saved PSW, CPU timer, clock comparator, prefix, and control, floating-point-control, and TOD programmable registers. Initial CPU reset sets the architectural mode to the ESA/390 mode if it is caused by activation of the load-normal key.

Subsystem reset provides a means for clearing floating interruption conditions as well as for invoking I/O-system reset.

Clear reset causes initial CPU reset and subsystem reset to be performed and, additionally, clears or initializes all storage locations and registers in all CPUs in the configuration, with the exception of the TOD clock. Such clearing is useful in debugging programs and in ensuring user privacy. Clear reset also releases all locks used by the PERFORM LOCKED OPERATION instruction. Clear reset sets the architectural mode to the ESA/390 mode. Clearing does not affect external storage, such as direct-access storage devices used by the control program to hold the contents of unaddressable pages.

CPU power-on reset causes initial CPU reset to be performed and clears the contents of general registers, access registers, and floating-point registers to zeros with valid checking-block code. Locks used by PERFORM LOCKED OPERATION and associated with the CPU are released unless they are held by a CPU already powered on. The power-on-reset sequences for the TOD clock, main storage, and the channel subsystem may be included as part of the CPU power-on sequence, or the power-on sequence for these units may be initiated separately. If CPU power-on reset establishes the configuration, it sets the architectural mode to the ESA/390 mode; otherwise, it sets the architectural mode to that of the CPUs already in the configuration.

CPU reset, initial CPU reset, subsystem reset, and clear reset may be initiated manually by using the operator facilities (see Chapter 12, “Operator Facilities”). Initial CPU reset is part of the initial-program-loading function. Figure 4-10 on page 4-48 summarizes how these four resets are manually initiated. Power-on reset is performed as part of turning power on. The reset actions are tabulated in Figure 4-11 on page 4-49. For information concerning which resets can be performed by the SIGNAL PROCESSOR instruction, see “Signal-Processor Orders” on page 4-56.
<table>
<thead>
<tr>
<th>Key Activated</th>
<th>Function Performed on¹</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU on Which Key Was Activated</td>
</tr>
<tr>
<td>System-reset-normal key</td>
<td>CPU reset</td>
</tr>
<tr>
<td>System-reset-clear key</td>
<td>Clear reset²</td>
</tr>
<tr>
<td>Load-normal key</td>
<td>Initial CPU reset, followed by IPL</td>
</tr>
<tr>
<td>Load-clear key</td>
<td>Clear reset², followed by IPL</td>
</tr>
</tbody>
</table>

**Explanation:**

¹ Activation of a system-reset or load key may change the configuration, including the connection with I/O, storage units, and other CPUs.

² Only the CPU elements of this reset apply.

³ Only the non-CPU elements of this reset apply.

*Figure 4-10. Manual Initiation of Resets*
<table>
<thead>
<tr>
<th>Area Affected</th>
<th>Sub-system Reset</th>
<th>CPU Reset</th>
<th>Initial CPU Reset</th>
<th>Clear Reset</th>
<th>Power-On Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>U</td>
<td>S</td>
<td>S(^1)</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>PSW</td>
<td>U</td>
<td>U/V#</td>
<td>C(^*1)</td>
<td>C(^*1)</td>
<td>C(^*1)</td>
</tr>
<tr>
<td>Saved PSW for use by SIGNAL</td>
<td>U</td>
<td>U/sv</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>PROCESSOR set-architecture order</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix</td>
<td>U</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>CPU timer</td>
<td>U</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Clock comparator</td>
<td>U</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>TOD programmable register</td>
<td>U</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Control registers</td>
<td>U</td>
<td>U/V</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Floating-point-control register</td>
<td>U</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Access registers</td>
<td>U</td>
<td>U/V</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>General registers</td>
<td>U</td>
<td>U/V</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Floating-point registers</td>
<td>U</td>
<td>U/V</td>
<td>U/V</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Storage keys</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>C</td>
<td>C(^2)</td>
</tr>
<tr>
<td>Volatile main storage</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>C</td>
<td>C(^2)</td>
</tr>
<tr>
<td>Nonvolatile main storage</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Expanded storage</td>
<td>U(^3)</td>
<td>U(^3)</td>
<td>U(^3)</td>
<td>U(^3)</td>
<td>C(^2)</td>
</tr>
<tr>
<td>TOD clock</td>
<td>U(^4)</td>
<td>U(^4)</td>
<td>U(^4)</td>
<td>T(^2)</td>
<td></td>
</tr>
<tr>
<td>Floating interruption conditions</td>
<td>C</td>
<td>U</td>
<td>U</td>
<td>C</td>
<td>C(^2)</td>
</tr>
<tr>
<td>I/O system</td>
<td>R</td>
<td>U</td>
<td>U</td>
<td>R</td>
<td>R(^5)</td>
</tr>
<tr>
<td>PERFORM LOCKED OPERATION locks</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>RC</td>
<td>RP</td>
</tr>
</tbody>
</table>

**Explanation:**

If the architectural mode is changed from Z/Architecture to ESA/390 (the reset is due to activation of the load-normal key on another CPU), the 16-byte PSW first is saved, for use by the SIGNAL PROCESSOR set-architecture order, and then does not remain unchanged. Instead, it is changed to an eight-byte PSW, and the bits of the eight-byte PSW are set as follows. Bits 0-11 and 13-32 are set equal to the same bits of the 16-byte PSW, bit 12 is set to one, and bits 33-63 are set equal to bits 97-127 of the 16-byte PSW. The PSW is invalid in the ESA/390 mode if PSW bit 31 is one.

* Clearing the contents of the PSW to zero causes the PSW to be invalid if the architectural mode is ESA/390.

1 When the IPL sequence follows the reset function on that CPU, the CPU does not necessarily enter the stopped state, and the PSW is not necessarily cleared to zeros.

2 When these units are separately powered, the action is performed only when the power for the unit is turned on.

Figure 4-11 (Part 1 of 3). Summary of Reset Actions
Explanation (Continued):

3 Access to change expanded storage at the time a reset function is performed may cause the contents of the 4K-byte block in expanded storage to be unpredictable. Access to examine expanded storage does not affect the contents of the expanded storage.

4 Access to the TOD clock by means of STORE CLOCK at the time a reset function is performed does not cause the value of the TOD clock to be affected.

5 When the channel subsystem is separately powered or consists of multiple elements which are separately powered, the reset action is applied only to those subchannels, channel paths, and I/O control units and devices on those paths associated with the element which is being powered on.

C The condition or contents are cleared. If the area affected is a field, the contents are set to zero with valid checking-block code.

I The state or contents are initialized. If the area affected is a field, the contents are set to the initial value with valid checking-block code.

R I/O-system reset is performed in the channel subsystem. As part of this reset, system reset is signaled to all I/O control units and devices attached to the channel subsystem.

RC All locks in the configuration are released.

RP All locks in the configuration are released except for ones held by CPUs already powered on.

S The CPU is reset; current operations, if any, are terminated; the ALB and TLB are cleared of entries; interruption conditions in the CPU are cleared; and the CPU is placed in the stopped state. The effect of performing the start function is unpredictable when the stopped state has been entered by means of a reset. If the reset is initiated by the system-reset-clear, load-normal, or load-clear key or by a CPU power-on reset that establishes the configuration, the architectural mode is set to the ESA/390 mode; otherwise, the architectural mode is unchanged, except that power-on reset sets the mode to that of the CPUs already in the configuration.
Explanation (Continued):

T The TOD clock is initialized to zero and validated; it enters the not-set state.

U The state, condition, or contents of the field remain unchanged. However, the result is unpredictable if an operation is in progress that changes the state, condition, or contents of the field at the time of reset.

U/sv The saved PSW remains unchanged if the reset is due to activation of the system-reset-normal key or the SIGNAL PROCESSOR CPU-reset order, or it is set with the value of the current 16-byte PSW if the reset is due to activation of the load-normal key.

U/V The contents remain unchanged, provided the field is not being changed at the time the reset function is performed. However, on some models the checking-block code of the contents may be made valid. The result is unpredictable if an operation is in progress that changes the contents of the field at the time of reset.

Figure 4-11 (Part 3 of 3). Summary of Reset Actions

CPU Reset

CPU reset causes the following actions:

1. The execution of the current instruction or other processing sequence, such as an interruption, is terminated, and all program-interruption and supervisor-call-interruption conditions are cleared.

2. Any pending external-interruption conditions which are local to the CPU are cleared. Floating external-interruption conditions are not cleared.

3. Any pending machine-check-interruption conditions and error indications which are local to the CPU and any check-stop states are cleared. Floating machine-check-interruption conditions are not cleared. Any machine-check condition which is reported to all CPUs in the configuration and which has been made pending to a CPU is said to be local to the CPU.

4. All copies of prefetched instructions or operands are cleared. Additionally, any results to be stored because of the execution of instructions in the current checkpoint interval are cleared.

5. The ART-lookaside buffer and translation-lookaside buffer are cleared of entries.

6. If the reset is caused by activation of the load-normal key on any CPU in the configuration, the following actions occur:
   a. The architectural mode of the CPU (and of all other CPUs in the configuration because of the initial CPU reset or CPU resets performed by them) is changed from the z/Architecture mode to the ESA/390 mode.
   b. The current PSW is saved for subsequent use by a SIGNAL PROCESSOR set-architecture order that restores the z/Architecture mode.
   c. The current PSW is changed from 16 bytes to eight bytes. The bits of the eight-byte PSW are set as follows: bits 0-11 and 13-32 are set equal to the same bits of the 16-byte PSW, bit 12 is set to one, and bits 33-63 are set equal to bits 97-127 of the 16-byte PSW.

A CPU reset caused by activation of the system-reset-normal key or by the SIGNAL PROCESSOR CPU-reset order, and any CPU reset in the ESA/390 mode, do not affect the saved z/Architecture PSW.

7. The CPU is placed in the stopped state after actions 1-6 have been completed. When the CCW-type IPL sequence follows the reset function on that CPU, the CPU enters the load
state at the completion of the reset function and does not necessarily enter the stopped state during the execution of the reset operation. When the list-directed IPL sequence follows the reset function on that CPU, the CPU enters the operating state and does not necessarily enter the stopped state during the execution of the reset operation.

Registers, storage contents, and the state of conditions external to the CPU remain unchanged by CPU reset. However, the subsequent contents of the register, location, or state are unpredictable if an operation is in progress that changes the contents at the time of the reset. A lock held by the CPU when executing PERFORM LOCKED OPERATION is not released by CPU reset.

When the reset function in the CPU is initiated at the time the CPU is executing an I/O instruction or is performing an I/O interruption, the current operation between the CPU and the channel subsystem may or may not be completed, and the resultant state of the associated channel-subsystem facility may be unpredictable.

Programming Notes:
1. Most operations which would change a state, a condition, or the contents of a field cannot occur when the CPU is in the stopped state. However, some signal-processor functions and some operator functions may change these fields. To eliminate the possibility of losing a field when CPU reset is issued, the CPU should be stopped, and no operator functions should be in progress.
2. If the architectural mode is changed to the ESA/390 mode and bit 31 of the current PSW is one, the PSW is invalid.

Initial CPU Reset
Initial CPU reset combines the CPU reset functions with the following clearing and initializing functions:

1. If the reset is caused by activation of the load-normal key, the architectural mode of the CPU (and of all other CPUs in the configuration) is set to the ESA/390 mode.

2. The contents of the current PSW, saved PSW (for use by the set-architecture order of SIGNAL PROCESSOR), prefix, CPU timer, clock comparator, and TOD programmable register are set to zero. When the IPL sequence follows the reset function on that CPU, the contents of the PSW are not necessarily set to zero.

3. The contents of the control registers are set to their initial z/Architecture values. All 64 bits of the control registers are set regardless of whether the CPU is in the ESA/390 or the z/Architecture architectural mode.

4. The contents of the floating-point-control register are set to zero.

These clearing and initializing functions include validation.

Setting the current PSW to zero when the CPU is in the ESA/390 architectural mode at the end of the operation causes the PSW to be invalid, since PSW bit 12 must be one in that mode. Thus, in this case if the CPU is placed in the operating state after a reset without first introducing a new PSW, a specification exception is recognized.

Subsystem Reset
Subsystem reset operates only on those elements in the configuration which are not CPUs. It performs the following actions:

1. I/O-system reset is performed by the channel subsystem (see "I/O-System Reset" on page 17-13).

2. All floating interruption conditions in the configuration are cleared.

As part of I/O-system reset, pending I/O-interruption conditions are cleared, and system reset is signaled to all control units and devices attached to the channel subsystem (see "I/O-System Reset" on page 17-13). The effect of system reset on I/O control units and devices and the resultant control-unit and device state are described in the appropriate System Library publication for the control unit or device. A system reset, in general, resets only those functions in a shared control unit or device that are associated with the particular channel path signaling the reset.
Clear Reset
Clear reset combines the initial-CPU-reset function with an initializing function which causes the following actions:

1. The architectural mode of all CPUs in the configuration is set to the ESA/390 mode.
2. The access, general, and floating-point registers of all CPUs in the configuration are set to zero. All 64 bits of the general registers are set to zero regardless of whether the CPU was in the ESA/390 or z/Architecture architectural mode when the clear-reset function was initiated.
3. The contents of the main storage in the configuration and the associated storage keys are set to zero with valid checking-block code.
4. The locks used by any CPU in the configuration when executing the PERFORM LOCKED OPERATION instruction are released.
5. A subsystem reset is performed.

Validation is included in setting registers and in clearing storage and storage keys.

Programming Notes:

1. The architectural mode is not changed by activation of the system-reset-normal key or by execution of a SIGNAL PROCESSOR CPU-reset or initial-CPU-reset order. All CPUs in the configuration are always in the same architectural mode.
2. For the CPU-reset operation not to affect the contents of fields that are to be left unchanged, the CPU must not be executing instructions and must be disabled for all interruptions at the time of the reset. Except for the operation of the CPU timer and for the possibility of a machine-check interruption occurring, all CPU activity can be stopped by placing the CPU in the wait state and by disabling it for I/O and external interruptions. To avoid the possibility of causing a reset at the time that the CPU timer is being updated or a machine-check interruption occurs, the CPU must be in the stopped state.
3. CPU reset, initial CPU reset, subsystem reset, and clear reset do not affect the value and state of the TOD clock.
4. The conditions under which the CPU enters the check-stop state are model-dependent and include malfunctions that preclude the completion of the current operation. Hence, if CPU reset or initial CPU reset is executed while the CPU is in the check-stop state, the contents of the PSW, registers, and storage locations, including the storage keys and the storage location accessed at the time of the error, may have unpredictable values, and, in some cases, the contents may still be in error after the check-stop state is cleared by these resets. In this situation, a clear reset is required to clear the error.

Power-On Reset
The power-on-reset function for a component of the machine is performed as part of the power-on sequence for that component.

The power-on sequences for the TOD clock, main storage, expanded storage, and channel subsystem may be included as part of the CPU power-on sequence, or the power-on sequence for these units may be initiated separately. The following sections describe the power-on resets for the CPU, TOD clock, main storage, expanded storage, and channel subsystem. See also [Chapter 17, “I/O Support Functions,” and the appropriate System Library publication for the channel subsystem, control units, and I/O devices.]

CPU Power-On Reset: The power-on reset causes initial CPU reset to be performed and may or may not cause I/O-system reset to be performed in the channel subsystem. The contents of general registers, access registers, and floating-point registers are cleared to zeros with valid checking-block code. Locks used by PERFORM LOCKED OPERATION and associated with the CPU are released unless they are held by a CPU already powered on. If the reset is associated with establishing a configuration, the CPU is placed in the ESA/390 mode; otherwise, the CPU is placed in the architectural mode of the CPUs already in the configuration.

TOD-Clock Power-On Reset: The power-on reset causes the value of the TOD clock to be set to zero with valid checking-block code and causes the clock to enter the not-set state.

Main-Storage Power-On Reset: For volatile main storage (one that does not preserve its contents when power is off) and for storage keys, power-on reset causes zeros with valid checking-
block code to be placed in these fields. The contents of nonvolatile main storage, including the checking-block code, remain unchanged.

**Expanded-Storage Power-On Reset:** The contents of expanded storage are cleared to zeros with valid checking-block code.

**Channel-Subsystem Power-On Reset:** The channel-subsystem power-on reset causes I/O-system reset to be performed in the channel subsystem. (See "I/O-System Reset" on page 17-13.)

## Initial Program Loading

Initial program loading (IPL) provides a manual means for causing a program to be read from a designated device and for initiating execution of that program.

Some models may provide additional controls and indications relating to IPL; this additional information is specified in the System Library publication for the model.

There are two types of IPL: CCW-type IPL and list-directed IPL. CCW-type IPL is provided by all machine configurations. List-directed IPL may be provided, depending on the model.

CCW-type IPL is described below. List-directed IPL is described in "List-Directed IPL" on page 17-20.

### CCW-Type IPL

CCW-type IPL is initiated manually by setting the load-unit-address controls to a four-digit number to designate an input device and by subsequently activating the load-clear or load-normal key for a particular CPU. In the description which follows, the term “this CPU” refers to the CPU in the configuration for which the load-clear or load-normal key was activated.

Activating the load-clear key causes a clear reset to be performed on the configuration.

Activating the load-normal key causes an initial CPU reset to be performed on this CPU, CPU reset to be propagated to all other CPUs in the configuration, and a subsystem reset to be performed on the remainder of the configuration.

Activating the load-clear key or the load-normal key sets the architectural mode to the ESA/390 mode. For ease of reference, the additional elements of the description of ESA/390 initial program loading are given below.

In the loading part of the operation, after the resets have been performed, this CPU then enters the load state. This CPU does not necessarily enter the stopped state during the execution of the reset operations. The load indicator is on while the CPU is in the load state.

Subsequently, a channel-program read operation is initiated from the I/O device designated by the load-unit-address controls. The effect of executing the channel program is as if a format-0 CCW beginning at absolute storage location 0 specified a read command with the modifier bits zeros, a data address of zero, a byte count of 24, the chain-command and SLI flags ones, and all other flags zeros.

The details of the channel-subsystem portion of the IPL operation are defined in "Initial Program Loading" on page 17-17.

When the IPL I/O operation is completed successfully, the subsystem-identification word for the IPL device is stored in absolute storage locations 184-187, zeros are stored in absolute storage locations 188-191, and a new PSW is loaded from absolute storage locations 0-7. If the PSW loading is successful and no machine malfunctions are detected, this CPU leaves the load state, and the load indicator is turned off. If the rate control is set to the process position, the CPU enters the operating state, and the CPU operation proceeds under control of the new PSW. If the rate control is set to the instruction-step position, the CPU enters the stopped state, with the manual indicator on, after the new PSW is loaded.

If the IPL I/O operation or the PSW loading is not completed successfully, the CPU remains in the load state, and the load indicator remains on. The contents of absolute storage locations 0-7 are unpredictable.
Store Status

The store-status operation places an architectural-mode identification and the contents of the CPU registers, except for the TOD clock, in assigned storage locations.

Figure 4-12 on page 4-55 lists the fields that are stored, their length, and their location in main storage.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length in Bytes</th>
<th>Absolute Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural-mode id</td>
<td>1</td>
<td>163</td>
</tr>
<tr>
<td>Fl-pt registers 0-15</td>
<td>128</td>
<td>4608</td>
</tr>
<tr>
<td>General registers 0-15</td>
<td>128</td>
<td>4736</td>
</tr>
<tr>
<td>Current PSW</td>
<td>16</td>
<td>4864</td>
</tr>
<tr>
<td>Prefix</td>
<td>4</td>
<td>4888</td>
</tr>
<tr>
<td>Fl-pt control register</td>
<td>4</td>
<td>4892</td>
</tr>
<tr>
<td>TOD programmable register</td>
<td>4</td>
<td>4900</td>
</tr>
<tr>
<td>CPU timer</td>
<td>8</td>
<td>4904</td>
</tr>
<tr>
<td>Zeros</td>
<td>1</td>
<td>4912</td>
</tr>
<tr>
<td>Bits 0-55 of clock comparator</td>
<td>7</td>
<td>4913</td>
</tr>
<tr>
<td>Access registers 0-15</td>
<td>64</td>
<td>4928</td>
</tr>
<tr>
<td>Control registers 0-15</td>
<td>128</td>
<td>4992</td>
</tr>
</tbody>
</table>

Figure 4-12. Assigned Storage Locations for Store Status

During the execution of the store-status operation, zeros are stored in bit positions 0-6, and a one is stored in bit position 7, of absolute location 163, the store-status architectural-mode identification.

When the CPU is in the ESA/390 architectural mode, the store-status operation stores all zeros at absolute location 163.

When bits 0-55 of the clock comparator are stored beginning at absolute location 4913, zeros are stored at absolute location 4912.

The contents of the registers are not changed. If an error is encountered during the operation, the CPU enters the check-stop state.

The store-status operation can be initiated manually by use of the store-status key (see Chapter 12, “Operator Facilities”). The store-status operation can also be initiated at the addressed CPU by executing SIGNAL PROCESSOR, specifying the stop-and-store-status order. Execution of SIGNAL PROCESSOR specifying the store-status-at-address order permits the same status information, except for the store-status architectural-mode identification, to be stored at a designated address (see Signal-Processor Orders on page 4-56).

Multiprocessing

The multiprocessing facility provides for the interconnection of CPUs, via a common main storage, in order to enhance system availability and to share data and resources. The multiprocessing facility includes the following facilities:

- Shared main storage
- CPU-to-CPU interconnection
- TOD-clock synchronization

Associated with these facilities is an external-interruption condition (malfunction alert), which is described in Chapter 6, “Interruptions” and control-register positions for the TOD-clock-sync-control bit and for the mask for the external-interruption condition, which are listed in Control Registers on page 4-7.

The channel subsystem, including all subchannels, in a multiprocessing configuration can be accessed by all CPUs in the configuration. I/O-interruption conditions are floating and can be accepted by any CPU in the configuration.

Shared Main Storage

The shared-main-storage facility permits more than one CPU to have access to common main-storage locations. All CPUs having access to a common main-storage location have access to the entire 4K-byte block containing that location and to the associated storage key. The channel subsystem and all CPUs in the configuration refer to a shared main-storage location using the same absolute address.

CPU-Address Identification

Each CPU has a number assigned, called its CPU address. A CPU address uniquely identifies one CPU within a configuration. The CPU is designated by specifying this address in the CPU-address field of SIGNAL PROCESSOR. The CPU signaling a malfunction alert, emergency signal, or external call is identified by storing this address in the CPU-address field with the interruption. The CPU address is assigned during
system installation and is not changed as a result of reconfiguration changes. The program can determine the address of the CPU by using STORE CPU ADDRESS.

**CPU Signaling and Response**

The CPU-signaling-and-response facility consists of SIGNAL PROCESSOR and a mechanism to interpret and act on several order codes. The facility provides for communications among CPUs, including transmitting, receiving, and decoding a set of assigned order codes; initiating the specified operation; and responding to the signaling CPU. A CPU can address SIGNAL PROCESSOR to itself. SIGNAL PROCESSOR is described in Chapter 10, "Control Instructions."

**Signal-Processor Orders**

The signal-processor orders are specified in bit positions 56-63 of the second-operand address of SIGNAL PROCESSOR and are encoded as shown in Figure 4-13.

<table>
<thead>
<tr>
<th>Code (Hex)</th>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Unassigned</td>
</tr>
<tr>
<td>01</td>
<td>Sense</td>
</tr>
<tr>
<td>02</td>
<td>External call</td>
</tr>
<tr>
<td>03</td>
<td>Emergency signal</td>
</tr>
<tr>
<td>04</td>
<td>Start</td>
</tr>
<tr>
<td>05</td>
<td>Stop</td>
</tr>
<tr>
<td>06</td>
<td>Restart</td>
</tr>
<tr>
<td>07</td>
<td>Unassigned</td>
</tr>
<tr>
<td>08</td>
<td>Unassigned</td>
</tr>
<tr>
<td>09</td>
<td>Stop and store status</td>
</tr>
<tr>
<td>0A</td>
<td>Unassigned</td>
</tr>
<tr>
<td>0B</td>
<td>Initial CPU reset</td>
</tr>
<tr>
<td>0C</td>
<td>CPU reset</td>
</tr>
<tr>
<td>0D</td>
<td>Set prefix</td>
</tr>
<tr>
<td>0E</td>
<td>Store status at address</td>
</tr>
<tr>
<td>0F-11</td>
<td>Unassigned</td>
</tr>
<tr>
<td>12</td>
<td>Set architecture</td>
</tr>
<tr>
<td>13-FF</td>
<td>Unassigned</td>
</tr>
</tbody>
</table>

*Figure 4-13. Encoding of Orders*

The orders are defined as follows:

**Sense:** The addressed CPU presents its status to the issuing CPU (see "Status Bits" on page 4-61 for a definition of the bits). No other action is caused at the addressed CPU. The status, if not all zeros, is stored in the general register designated by the R1 field of the SIGNAL PROCESSOR instruction, and condition code 1 is set; if all status bits are zeros, condition code 0 is set.

**External Call:** An external-call external-interruption condition is generated at the addressed CPU. The interruption condition becomes pending during the execution of SIGNAL PROCESSOR. The associated interruption occurs when the CPU is enabled for that condition and does not necessarily occur during the execution of SIGNAL PROCESSOR. The address of the CPU sending the signal is provided with the interruption code when the interruption occurs. Only one external-call condition can be kept pending in a CPU at a time. The order is accepted only when the addressed CPU is in the stopped or the operating state.

**Emergency Signal:** An emergency-signal external-interruption condition is generated at the addressed CPU. The interruption condition becomes pending during the execution of SIGNAL PROCESSOR. The associated interruption occurs when the CPU is enabled for that condition and does not necessarily occur during the execution of SIGNAL PROCESSOR. The address of the CPU sending the signal is provided with the interruption code when the interruption occurs. At any one time the receiving CPU can keep pending one emergency-signal condition for each CPU in the configuration, including the receiving CPU itself. The order is accepted only when the addressed CPU is in the stopped or the operating state.

**Start:** The addressed CPU performs the start function (see "Stopped, Operating, Load, and Check-Stop States" on page 4-1). The CPU does not necessarily enter the operating state during the execution of SIGNAL PROCESSOR. The order is effective only when the addressed CPU is in the stopped state. The effect of performing the start function is unpredictable when the stopped state has been entered by reset.

**Stop:** The addressed CPU performs the stop function (see "Stopped, Operating, Load, and Check-Stop States" on page 4-1). The CPU does not necessarily enter the stopped state during the execution of SIGNAL PROCESSOR. The order is effective only when the CPU is in the operating state.
**Restart:** The addressed CPU performs the restart operation (see “Restart Interruption” on page 6-47). The CPU does not necessarily perform the operation during the execution of SIGNAL PROCESSOR. The order is effective only when the addressed CPU is in the stopped or the operating state.

**Stop and Store Status:** The addressed CPU performs the stop function, followed by the store-status operation (see “Store Status” on page 4-55). The CPU does not necessarily complete the operation, or even enter the stopped state, during the execution of SIGNAL PROCESSOR. The order is effective only when the addressed CPU is in the stopped or the operating state.

**Initial CPU Reset:** The addressed CPU performs initial CPU reset (see “Resets” on page 4-47). The execution of the reset does not affect the architectural mode or other CPUs and does not cause I/O to be reset. The reset operation is not necessarily completed during the execution of SIGNAL PROCESSOR.

**CPU Reset:** The addressed CPU performs CPU reset (see “Resets” on page 4-47). The execution of the reset does not affect the architectural mode or other CPUs and does not cause I/O to be reset. The reset operation is not necessarily completed during the execution of SIGNAL PROCESSOR.

**Set Prefix:** The contents of bit positions 33-50 of the parameter register of the SIGNAL PROCESSOR instruction are treated as a prefix value, which replaces bits 33-50 of the prefix register of the addressed CPU. Bits 0-32 and 51-63 of the parameter register are ignored. The order is accepted only if the addressed CPU is in the stopped state, the value to be placed in the prefix register designates an 8K block which is available in the configuration, and no other condition precludes accepting the order. Verification of the stopped state of the addressed CPU and of the availability of the designated storage is performed during execution of SIGNAL PROCESSOR. If accepted, the order is not necessarily completed during the execution of SIGNAL PROCESSOR.

The parameter register has the following format:

```
+-----------------+-----------------+-----------------+
|  Prefix Value   |  Prefix Value   |  Prefix Value   |
+-----------------+-----------------+-----------------+
| 0   33  51  63   | 0   33  51  63   | 0   33  51  63   |
```

The set-prefix order is completed as follows:

- If the addressed CPU is not in the stopped state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.

- The value to be placed in the prefix register of the addressed CPU is tested for the availability of the designated storage. The absolute address of an 8K-byte area of storage is formed by appending 13 zeros to the right and 33 zeros to the left of bits 33-50 of the parameter value. This address is treated as a 64-bit absolute address regardless of whether the sending and receiving CPUs are in the 24-bit, 31-bit, or 64-bit addressing mode. The two 4K-byte blocks of storage within the new prefix area are accessed. The accesses to the blocks are not subject to protection, and the associated reference bits may or may not be set to one. If either block is not available in the configuration, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.

- The value is placed in the prefix register of the addressed CPU.

- The ALB and TLB of the addressed CPU are cleared of their contents.

- A serializing and checkpoint-synchronizing function is performed on the addressed CPU following insertion of the new prefix value.

**Store Status at Address:** The contents of bit positions 33-54 of the parameter register of the SIGNAL PROCESSOR instruction are used as the origin of a 512-byte area on a 512-byte boundary in absolute storage into which the status of the addressed CPU is stored. Bits 0-32 and 55-63 of the parameter register are ignored.

The order is accepted only if the addressed CPU is in the stopped state, the status-area origin designates a location which is available in the config-
uration, and no other condition precludes accepting the order. Verification of the stopped state of the addressed CPU and of the availability of the designated storage is performed during execution of SIGNAL PROCESSOR. If accepted, the order is not necessarily completed during the execution of SIGNAL PROCESSOR.

The parameter register has the following format:

```
|----| Status-Area Origin |----|
```

The store-status-at-address order is completed as follows:

- If the addressed CPU is not in the stopped state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R₁ field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.

- The address of the area into which status is to be stored is tested for availability. The absolute address of a 512-byte area of storage is formed by appending 9 zeros to the right and 33 zeros to the left of bits 33-54 of the parameter value. This address is treated as a 64-bit absolute address regardless of whether the sending and receiving CPUs are in the 24-bit, 31-bit, or 64-bit addressing mode. The 512-byte block of storage at this address is accessed. The access is not subject to protection, and the associated reference bit may or may not be set to one. If the block is not available in the configuration, the order is not accepted by the addressed CPU, bit 55 (invalid parameter) of the general register designated by the R₁ field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.

- The status of the addressed CPU is placed in the designated area. The information stored, and the format of the area receiving the information, are the same as for the stop-and-store-status order, except that each field, rather than being stored at an offset from the beginning of absolute storage, is stored in the designated area at the offsets listed in Figure 4-14, and except that an architectural-mode identification is not stored. Bytes 288-291 and 312-319 of the designated area remain unchanged.

- A serialization and checkpoint-synchronization function is performed on the addressed CPU following storing of the status.

<table>
<thead>
<tr>
<th>Field</th>
<th>Length in Bytes</th>
<th>Offset in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fl-pt registers 0-15</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>General registers 0-15</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Current PSW</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td>Prefix</td>
<td>4</td>
<td>280</td>
</tr>
<tr>
<td>Fl-pt-control register</td>
<td>4</td>
<td>284</td>
</tr>
<tr>
<td>TOD programmable register</td>
<td>4</td>
<td>292</td>
</tr>
<tr>
<td>CPU timer</td>
<td>8</td>
<td>296</td>
</tr>
<tr>
<td>Zeros</td>
<td>1</td>
<td>304</td>
</tr>
<tr>
<td>Bits 0-55 of clock comparator</td>
<td>7</td>
<td>305</td>
</tr>
<tr>
<td>Access registers 0-15</td>
<td>64</td>
<td>320</td>
</tr>
<tr>
<td>Control registers 0-15</td>
<td>128</td>
<td>384</td>
</tr>
</tbody>
</table>

![Figure 4-14. Location of Status Fields in Designated Area](image)

**Programming Note:** The architectural mode of the CPU that stored status in a designated area normally is indicated by bit 12 of the PSW stored at offset 256 in the area. The PSW is stored at the same offset, 256, in both the ESA/390 mode and the z/Architecture mode. Bit 12 is one in an ESA/390 PSW and zero in a z/Architecture PSW. The store-status-at-address order does not store the architectural-mode identification that is stored at absolute location 163 by the store-status operation and the stop-and-store-status order.

**Set Architecture:** The contents of bit positions 56-63 of the parameter register are used as a code specifying an architectural mode to which all CPUs in the configuration are to be set: code 0 specifies the ESA/390 mode, and codes 1 and 2 specify the z/Architecture mode. Code 1 specifies that, for each of all CPUs in the configuration, the current ESA/390 PSW is to be transformed to a z/Architecture PSW. Code 2 specifies that the PSW of the CPU executing SIGNAL PROCESSOR is to be transformed to a z/Architecture PSW and that, for each of all other CPUs in the configuration, the PSW is to be set with the value of the saved PSW for that CPU. The setting of the PSW with the value of the saved PSW will restore the PSW that existed when the CPU was last in the z/Architecture mode, provided that the saved PSW has not been set to all zeros by a reset.
Bits 0-55 of the parameter register are ignored. The contents of the CPU-address register of the SIGNAL PROCESSOR instruction are ignored; all other CPUs in the configuration are considered to be addressed.

The order is accepted only if the code is 0, 1, or 2, the CPU is not already in the mode specified by the code, each of all other CPUs is in either the stopped or the check-stop state, and no other condition precludes accepting the order. If accepted, the order is completed by all CPUs during the execution of SIGNAL PROCESSOR. In no case can different CPUs be in different architectural modes.

The set-architecture order is completed as follows:

- If the code in the parameter register is not 0, 1, or 2, or if the CPU is already in the architectural mode specified by the code, the order is not accepted. Instead, bit 55 (invalid parameter) of the general register designated by the R1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- If it is not true that all other CPUs in the configuration are in the stopped or check-stop state, the order is not accepted. Instead, bit 54 (incorrect state) of the general register designated by the R1 field of the SIGNAL PROCESSOR instruction is set to one, and condition code 1 is set.
- The architectural mode of all CPUs in the configuration is set as specified by the code.
- If the order changes the architectural mode from ESA/390 to z/Architecture and the code is 1, then, for each CPU in the configuration, (1) the current PSW, which is the updated PSW in the case of the CPU executing SIGNAL PROCESSOR, is saved, and (2) the 16-byte current PSW is changed to an eight-byte PSW by setting the bits of the eight-byte PSW as follows: bits 0-11 and 13-32 are set equal to the same bits of the 16-byte PSW, bit 12 is set to one, and bits 33-63 are set equal to bits 97-127 of the 16-byte PSW. Bit 51 of the z/Architecture prefix, which becomes bit 19 of the ESA/390 prefix, remains unchanged.
- The ALBs and TLBs of all CPUs in the configuration are cleared of their contents.
- A serialization and checkpoint-synchronization function is performed on all CPUs in the configuration.

If the order changes the architectural mode from z/Architecture to ESA/390 and the SIGNAL PROCESSOR instruction causes occurrence of an instruction-fetching PER event, only the rightmost 31 bits of the address of the instruction are stored in the ESA/390 PER-address field.

Programming Notes:

1. If the set-architecture order changes the architectural mode from z/Architecture to ESA/390 and bit 31 of the PSW is one, the PSW is invalid.
2. For a discussion of the relative performance of the SIGNAL PROCESSOR orders, see the programming note following the instruction SIGNAL PROCESSOR in Chapter 10, "Control Instructions."

Conditions Determining Response
Conditions Precluding Interpretation of the Order Code

The following situations preclude the initiation of the order. The sequence in which the situations are listed is the order of priority for indicating concurrently existing situations:

1. The access path to the addressed CPU is busy because a concurrently executed SIGNAL PROCESSOR is using the CPU-signaling-and-response facility. The CPU which is concurrently executing the instruction can be any CPU in the configuration other than this CPU, and the CPU address can be any address, including that of this CPU or an invalid address. The order is rejected. Condition code 2 is set.

2. The addressed CPU is not operational; that is, it is not provided in the installation, it is not in the configuration, it is in any of certain customer-engineer test modes, or its power is off. The order is rejected. Condition code 3 is set. This condition cannot arise as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

3. One of the following conditions exists at the addressed CPU:
   a. A previously issued start, stop, restart, stop-and-store-status, set-prefix, or store-status-at-address order has been accepted by the addressed CPU, and execution of the function requested by the order has not yet been completed.
   b. A manual start, stop, restart, or store-status function has been initiated at the addressed CPU, and the function has not yet been completed. This condition cannot arise as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

   If the currently specified order is sense, external call, emergency signal, start, stop, restart, stop and store status, set prefix, store status at address, or set architecture, then the order is rejected, and condition code 2 is set.

   If the conditions described in items 3 and 4 exists, the addressed CPU is referred to as “busy.” Busy is not indicated if the addressed CPU is in the check-stop state or when the operator-intervening condition exists. A CPU-busy condition is normally of short duration; however, the conditions described in item 3 may last indefinitely because of a string of interruptions. In this situation, however, the CPU does not appear busy to any of the reset orders.

   If the currently specified order is one of the reset orders, or an unassigned or not-implemented order, either the order is rejected and condition code 2 is set or the order code is interpreted as described in "Status Bits" on page 4-61.

4. One of the following conditions exists at the addressed CPU:
   a. A previously issued initial-CPU-reset or CPU-reset order has been accepted by the addressed CPU, and execution of the function requested by the order has not yet been completed.
   b. A manual-reset function has been initiated at the addressed CPU, and the function has not yet been completed. This condition cannot arise as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

   If the currently specified order is sense, external call, emergency signal, start, stop, restart, stop and store status, set prefix, store status at address, or set architecture, then the order is rejected, and condition code 2 is set.

   If the conditions described in items 1 and 2 above do not apply and operator-intervening and receiver-check status conditions do not exist at the addressed CPU, reset orders may be accepted regardless of whether the addressed CPU has completed a previously accepted order. This may cause the previous order to be lost when it is only partially completed, making unpredictable whether the results defined for the lost order are obtained.
Status Bits
Various status conditions are defined whereby the issuing and addressed CPUs can indicate their responses to the specified order. The status conditions and their bit positions in the general register designated by the R1 field of the SIGNAL PROCESSOR instruction are shown in Figure 4-15 on page 4-61.

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Status Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Equipment check</td>
</tr>
<tr>
<td>33-53</td>
<td>Unassigned; zeros stored</td>
</tr>
<tr>
<td>54</td>
<td>Incorrect state</td>
</tr>
<tr>
<td>55</td>
<td>Invalid parameter</td>
</tr>
<tr>
<td>56</td>
<td>External-call pending</td>
</tr>
<tr>
<td>57</td>
<td>Stopped</td>
</tr>
<tr>
<td>58</td>
<td>Operator intervening</td>
</tr>
<tr>
<td>59</td>
<td>Check stop</td>
</tr>
<tr>
<td>60</td>
<td>Unassigned; zero stored</td>
</tr>
<tr>
<td>61</td>
<td>Inoperative</td>
</tr>
<tr>
<td>62</td>
<td>Invalid order</td>
</tr>
<tr>
<td>63</td>
<td>Receiver check</td>
</tr>
</tbody>
</table>

Figure 4-15. Status Conditions

The status condition assigned to bit position 32, and to bit position 55 when the order is set architecture, are generated by the CPU executing SIGNAL PROCESSOR. The remaining status conditions are generated by the addressed CPU.

When the invalid-parameter condition exists for the set-architecture order, bit 55 of the general register designated by the R1 field of the SIGNAL PROCESSOR instruction is set to one, all other bits in bit positions 32-63 are set to zeros, bits 0-31 of the register remain unchanged, and condition code 1 is set. No other action is taken.

When the equipment-check condition exists, except when the invalid-parameter condition exists for the set-architecture order, bit 32 of the general register designated by the R1 field of the SIGNAL PROCESSOR instruction is set to one, unassigned bits in bit positions 32-63 of the status register are set to zeros, the other status bits are unpredictable, and bits 0-31 of the register remain unchanged. In this case, condition code 1 is set independent of whether the access path to the addressed CPU is busy and independent of whether the addressed CPU is not operational, is busy, or has presented zero status.

When the access path to the addressed CPU is not busy and the addressed CPU is operational and does not indicate busy to the currently specified order, the addressed CPU presents its status to the issuing CPU. These status bits are of two types:

1. Status bits 54, 55 when the order is not set architecture, 56-59, and 61 indicate the presence of the corresponding conditions in the addressed CPU at the time the order code is received. Except in response to the sense order, each condition is indicated only when the condition precludes the successful execution of the specified order, although invalid parameter is not necessarily indicated when any other precluding condition exists. In the case of sense, all existing status conditions are indicated; the operator-intervening condition is indicated if it precludes the execution of any installed order.

2. Status bits 62 and 63 indicate that the corresponding conditions were detected by the addressed CPU during reception of the order.

If the presented status is all zeros, the addressed CPU has accepted the order, and condition code 0 is set at the issuing CPU; if the presented status is not all zeros, the order has been rejected, the status is stored at the issuing CPU in the general register designated by the R1 field of the SIGNAL PROCESSOR instruction, zeros are stored in the unassigned positions in bit positions 32-63 of the register, bits 0-31 of the register remain unchanged, and condition code 1 is set.

When the order is set architecture, “the addressed CPU” refers to each of the other CPUs in the configuration. Those CPUs, in an unpredictable order, are tested for a condition that causes setting of condition code 1, 2, or 3. Conditions are prioritized for a single CPU as if it were the only CPU addressed, but there is no prioritization across CPUs. If a condition is recognized, no further CPUs are tested, the condition code corresponding to the condition is set, and the execution of SIGNAL PROCESSOR is completed.

The status conditions are defined as follows:

**Equipment Check**: This condition exists when the CPU executing the instruction detects equipment malfunctioning that has affected only the execution of this instruction and the associated...
order. The order code may or may not have been transmitted and may or may not have been accepted, and the status bits provided by the addressed CPU may be in error. This condition is not detected if the invalid-parameter condition for the set-architecture order is detected.

**Incorrect State:** A set-prefix or store-status-at-address order has been rejected because the addressed CPU is not stopped, or a set-architecture order has been rejected because not all other CPUs are stopped or in the check-stop state. When applicable, this status is generated during execution of SIGNAL PROCESSOR and is indicated concurrently with other indications of conditions which preclude execution of the order, except that this status is not generated if an invalid-parameter condition exists for a set-architecture order.

**Invalid Parameter:** This condition exists in two cases:

1. The parameter value supplied with a set-prefix or store-status-at-address order designates a storage location which is not available in the configuration. When applicable, this status is generated during execution of SIGNAL PROCESSOR, except that it is not necessarily generated when another condition precluding execution of the order also exists.

2. The parameter value supplied with a set-architecture order either is not 0 or 1 or specifies the current architectural mode. When applicable, this status is generated during execution of SIGNAL PROCESSOR, and no other status is generated.

**External Call Pending:** This condition exists when an external-call interruption condition is pending in the addressed CPU because of a previously issued SIGNAL PROCESSOR order. The condition exists from the time an external-call order is accepted until the resultant external interruption has been completed or a CPU reset occurs. The condition may be due to the issuing CPU or another CPU. The condition, when present, is indicated only in response to sense and to external call.

**Stopped:** This condition exists when the addressed CPU is in the stopped state. The condition, when present, is indicated only in response to sense. This condition cannot be reported as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

**Operator Intervening:** This condition exists when the addressed CPU is executing certain operations initiated from local or remote operator facilities. The particular manually initiated operations that cause this condition to be present depend on the model and on the order specified. The operator-intervening condition may exist when the addressed CPU uses reloadable control storage to perform an order and the required licensed internal code has not been loaded by the IML function. The operator-intervening condition, when present, can be indicated in response to all orders. Operator intervening is indicated in response to sense if the condition is present and precludes the acceptance of any of the installed orders. The condition may also be indicated in response to unassigned or uninstalled orders. This condition cannot arise as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

**Check Stop:** This condition exists when the addressed CPU is in the check-stop state. The condition, when present, is indicated only in response to sense, external call, emergency signal, start, stop, restart, set prefix, store status at address, and stop and store status. The condition may also be indicated in response to unassigned or uninstalled orders. This condition cannot be reported as a result of a SIGNAL PROCESSOR instruction executed by a CPU addressing itself.

**Inoperative:** This condition indicates that the execution of the operation specified by the order code requires the use of a service processor which is inoperative. The failure of the service processor may have been previously reported by a service-processor-damage machine-check condition. The inoperative condition cannot occur for the emergency-signal, external-call, or sense order code.

**Invalid Order:** This condition exists during the communications associated with the execution of SIGNAL PROCESSOR when an unassigned or uninstalled order code is decoded.

**Receiver Check:** This condition exists when the addressed CPU detects malfunctioning of equipment during the communications associated with
the execution of SIGNAL PROCESSOR. When
this condition is indicated, the order has not been
initiated, and, since the malfunction may have
affected the generation of the remaining receiver
status bits, these bits are not necessarily valid. A
machine-check condition may or may not have
been generated at the addressed CPU.
The following chart summarizes which status conditions are presented to the issuing CPU in
response to each order code.
Status Condition
63
62
61
59
58
57
56
55
54

Receiver check! ────────────────────┐
Invalid order ────────────────────┐ │
Inoperative ────────────────────┐ │ │
Check stop ───────────────────┐ │ │ │
Operator intervening# ──────┐ │ │ │ │
Stopped ──────────────────┐ │ │ │ │ │
External call pending ──┐ │ │ │ │ │ │
Invalid parameter ────┐ │ │ │ │ │ │ │
Incorrect state ────┐ │ │ │ │ │ │ │ │
│ │ │ │ │ │ │ │ │
│ │ │ │ │ │ │ │ │
Order
│ │ │ │ │ │ │ │ │
        
Sense
  X X X X   X
External call
  X  X X   X
Emergency signal
    X X   X
Start
    X X X  X
Stop
    X X X  X
Restart
    X X X  X
Stop and store status     X X X  X
Initial CPU reset
    X  X  X
CPU reset
    X  X  X
Set prefix
X X   X X X  X
Store status at addr. X X   X X X  X
Set architecture
X X   X  X  X
Unassigned order
    X E X 1 X
Explanation:
#

The current state of the operator-intervening
condition may depend on the order code that
is being interpreted.

≠

If a one is presented in the receiver-check bit
position, the values presented in the other bit
positions are not necessarily valid.

0

A zero is presented in this bit position regardless of the current state of this condition.

1

A one is presented in this bit position.

X

A zero or a one is presented in this bit position, reflecting the current state of the corresponding condition.

E

Either a zero or the current state of the corresponding condition is indicated.

If the presented status bits are all zeros, the order
has been accepted, and the issuing CPU sets
condition code 0. If one or more ones are presented, the order has been rejected, and the
issuing CPU stores the status in the general register designated by the R field of the SIGNAL
PROCESSOR instruction and sets condition code
1.
Programming Notes:
1. All SIGNAL PROCESSOR orders except set
architecture (which in effect is addressed to all
other CPUs and affects all CPUs) can be
addressed to this same CPU. The following
are examples of functions obtained by a CPU
addressing SIGNAL PROCESSOR to itself:
a. Sense indicates whether an external-call
condition is pending.
b. External call and emergency signal cause
the corresponding interruption conditions
to be generated. External call can be
rejected because of a previously generated external-call condition.
c. Start sets condition code 0 and has no
other effect.
d. Stop causes the CPU to set condition
code 0, take pending interruptions for
which it is enabled, and enter the stopped
state.
e. Restart provides a means to store the
current PSW.
f. Stop and store status causes the machine
to stop and store all current status.
2. Two CPUs can simultaneously execute
SIGNAL PROCESSOR, with each CPU
addressing the other. When this occurs, one
CPU, but not both, can find the access path
busy because of the transmission of the order
code or status bits associated with SIGNAL
PROCESSOR that is being executed by the
other CPU. Alternatively, both CPUs can find
the access path available and transmit the
order codes to each other. In particular, two
CPUs can simultaneously stop, restart, or
reset each other.
3. To obtain status from another CPU which is in
the check-stop state by means of the storestatus-at-address order, a CPU reset operation should first be used to bring the CPU to
the stopped state. This reset order does not
Chapter 4. Control

4-63


alter the status, and, depending on the nature of the malfunction, provides the best chance of establishing conditions in the addressed CPU which allow status to be obtained.
Instructions

Each instruction consists of two major parts:

- An operation code (op code), which specifies the operation to be performed
- The designation of the operands that participate
Operands

Operands can be grouped in three classes: operands located in registers, immediate operands, and operands in storage. Operands may be either explicitly or implicitly designated.

Register operands can be located in general, floating-point, access, or control registers, with the type of register identified by the op code. The register containing the operand is specified by identifying the register in a four-bit field, called the R field, in the instruction. For some instructions, an operand is located in an implicitly designated register, the register being implied by the op code.

Immediate operands are contained within the instruction, and the 8-bit, 16-bit, or 32-bit field containing the immediate operand is called the I field.

Operands in storage may have an implied length; be specified by a bit mask; be specified by a four-bit or eight-bit length specification, called the L field, in the instruction; or have a length specified by the contents of a general register. The addresses of operands in storage are specified by means of a format that uses the contents of a general register as part of the address. This makes it possible to:

1. Specify a complete address by using an abbreviated notation
2. Perform address manipulation using instructions which employ general registers for operands
3. Modify addresses by program means without alteration of the instruction stream
4. Operate independent of the location of data areas by directly using addresses received from other programs

The address used to refer to storage either is contained in a register designated by the R field in the instruction or is calculated from a base address, index, and displacement, specified by the B, X, and D fields, respectively, in the instruction.

When the CPU is in the access-register mode, a B or R field may designate an access register in addition to being used to specify an address.

To describe the execution of instructions, operands are designated as first and second operands and, in some cases, third operands.

In general, two operands participate in an instruction execution, and the result replaces the first operand. However, CONVERT TO DECIMAL, TEST BLOCK, and instructions with "store" in the instruction name (other than STORE THEN AND SYSTEM MASK and STORE THEN OR SYSTEM MASK) use the second-operand address to designate a location in which to store. TEST AND SET, COMPARE AND SWAP, and COMPARE DOUBLE AND SWAP may perform an update on the second operand. Except when otherwise stated, the contents of all registers and storage locations participating in the addressing or execution part of an operation remain unchanged.

Instruction Formats

An instruction is one, two, or three halfwords in length and must be located in storage on a halfword boundary. Each instruction is in one of 21 basic formats: E, I, RI, RIE, RIL, RR, RRE, RRF, RS, RSI, RSL, RSY, RX, RXE, RXF, RXY, S, SI, SIY, SS, and SSE, with three variations of RRF, two of RI, RIL, RS, and RSY, and five of SS. See Figure 5-1.

<table>
<thead>
<tr>
<th>E Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
</tr>
<tr>
<td>0 15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>0 8 15</td>
</tr>
</tbody>
</table>

Figure 5-1 (Part 1 of 5). Basic Instruction Formats
RI Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>OpCd</th>
<th>I/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RIE Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>R/basesuperthree</th>
<th>I/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>40</td>
<td>47</td>
</tr>
</tbody>
</table>

RIL Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>OpCd</th>
<th>I/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RR Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>R/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

RRE Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>R/basesuperthree</th>
<th>R/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

RRF Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>R/basesuperthree</th>
<th>R/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>40</td>
<td>47</td>
</tr>
</tbody>
</table>

RS Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>B/basesupertwo</th>
<th>D/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

RSI Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R/basesuperone</th>
<th>R/basesuperthree</th>
<th>I/basesupertwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RSL Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L/basesuperone</th>
<th>B/basesuperone</th>
<th>D/basesuperone</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Figure 5-1 (Part 2 of 5). Basic Instruction Formats
### Instructions

Some instructions contain fields that vary slightly from the basic format, and in some instructions the operation performed does not follow the general rules stated in this section. All of these exceptions are explicitly identified in the individual instruction descriptions.

The format names indicate, in general terms, the classes of operands which participate in the operation and some details about fields:

- **E** denotes an operation using implied operands and an extended op-code field.
- **I** denotes an immediate operation.
- **RI** denotes a register-and-immediate operation and an extended op-code field.

### Instruction Formats

#### RSY Format

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>47</td>
</tr>
</tbody>
</table>

#### RX Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

#### RXE Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

#### RXF Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R3</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>31</td>
</tr>
</tbody>
</table>

#### RXY Format

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

#### SIY Format

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>47</td>
</tr>
</tbody>
</table>

#### SS Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
</tr>
</tbody>
</table>

#### SSE Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L1</th>
<th>L2</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

### Figure 5-1 (Part 4 of 5). Basic Instruction Formats

Instruction fields shown in Figure 5-1 on page 5-3 as containing slashes (/) are currently unassigned. These fields in an instruction should contain zeros; otherwise, the program may not operate compatibly in the future.

Some instructions contain fields that vary slightly from the basic format, and in some instructions the operation performed does not follow the general rules stated in this section. All of these exceptions are explicitly identified in the individual instruction descriptions.
RIE denotes a register-and-immediate operation and a longer extended op-code field.

RIL denotes a register-and-immediate operation, an extended op-code field, and a longer immediate field.

RR denotes a register-and-register operation.

RRE denotes a register-and-register operation and an extended op-code field.

RRF denotes a register-and-register operation, an extended op-code field, and an additional R field, M field, or both.

RS denotes a register-and-storage operation.

RSI denotes a register-and-immediate operation.

RSL denotes a storage operation (with an instruction format derived from the ESA/390 RSE format).

RSY denotes a register-and-storage operation, an extended op-code field, and a long displacement field.

RX denotes a register-and-indexed-storage operation.

RXE denotes a register-and-indexed-storage operation and an extended op-code field.

RXF denotes a register-and-indexed-storage operation, an extended op-code field, and an additional R field.

RXY denotes a register-and-indexed-storage operation, an extended op-code field, and a long displacement field.

S denotes an operation using an implied operand and storage.

SI denotes a storage-and-immediate operation.

SIY denotes a storage-and-immediate operation and a long displacement field.

SS denotes a storage-and-storage operation.

SSE denotes a storage-and-storage operation and an extended op-code field.

In the I, RR, RS, RSI, RX, SI, and SS formats, the first byte of an instruction contains the op code. In the E, RRE, RRF, R, RX, RXF, RXY, RS, RSY, RSI, RI, RIE, and RIL formats, the contents of the register designated by the R field are called the first operand. The register containing the first operand is sometimes referred to as the “first-operand location,” and sometimes as “register R1.” In the RR, RRE, and RRF formats, the R2 field designates the register containing the second operand, and the R2 field may designate the same register as R1. In the RRF, RXF, RS, RSY, RSI, and RIE formats, the use of the R3 field depends on the instruction. In the RS and RSY formats, the R3 field may instead be an M3 field specifying a mask.

The R field designates a general or access register in the general instructions, a general register in the control instructions, and a floating-point register in the floating-point instructions. However, in the instructions EXTRACT STACKED REGISTERS and LOAD ADDRESS EXTENDED, the R field designates both a general register and an access register, and, in the instructions LOAD CONTROL and STORE CONTROL, the R field designates a control register. (This paragraph refers only to register operands, not to the use of access registers in addressing storage operands.)

The first two bits of the first or only byte of the op code specify the length and format of the instruction, as follows:

<table>
<thead>
<tr>
<th>Bit Positions 0-1</th>
<th>Instruction Length (in Halfwords)</th>
<th>Instruction Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>One</td>
<td>I/E/RR</td>
</tr>
<tr>
<td>01</td>
<td>Two</td>
<td>RX</td>
</tr>
<tr>
<td>10</td>
<td>Two</td>
<td>RRE/RF/RX/RS/RSI/RI/S</td>
</tr>
<tr>
<td>11</td>
<td>Three</td>
<td>RXE/RXF/RX/RSY/RSI/RSL/RIE/RIL/SS/SSE/SIY</td>
</tr>
</tbody>
</table>

In the format illustration for each individual instruction description, the op-code field or fields show the op code as hexadecimal digits within single quotes. The hexadecimal representation uses 0-9 for the binary codes 0000-1001 and A-F for the binary codes 1010-1111.

The remaining fields in the format illustration for each instruction are designated by code names, consisting of one or two letters and possibly a subscript number. The subscript number denotes the operand to which the field applies.

Register Operands

In the RR, RRE, RRF, RX, RXE, RXF, RXY, RS, RSY, RSI, RI, RIE, and RIL formats, the contents of the register designated by the R field are called the first operand. The register containing the first operand is sometimes referred to as the “first-operand location,” and sometimes as “register R1.” In the RR, RRE, and RRF formats, the R2 field designates the register containing the second operand, and the R2 field may designate the same register as R1. In the RRF, RXF, RS, RSY, RSI, and RIE formats, the use of the R3 field depends on the instruction. In the RS and RSY formats, the R3 field may instead be an M3 field specifying a mask.

In the I, RR, RS, RSI, RX, SI, and SS formats, the first byte of an instruction contains the op code.
For access and floating-point registers, unless otherwise indicated in the individual instruction description, the register operand is one register in length (32 bits for an access register and 64 bits for a floating-point register), and the second operand is the same length as the first. For general and control registers, the register operand is in bit positions 32-63 of the 64-bit register or occupies the entire register, depending on the instruction.

Immediate Operands
In the I format, the contents of the eight-bit immediate-data field, the I field of the instruction, are directly used as the operand.

In the SI format, the contents of the eight-bit immediate-data field, the I2 field of the instruction, are used directly as the second operand. The B1 and D1 fields specify the first operand, which is one byte in length. In the SIY format, the operation is the same except that DH1 and DL1 fields are used instead of a D1 field.

In the RI format for the instructions ADD HALFWORD IMMEDIATE, COMPARE HALFWORD IMMEDIATE, LOAD HALFWORD IMMEDIATE, and MULTIPLY HALFWORD IMMEDIATE, the contents of the 16-bit I2 field of the instruction are used directly as a signed binary integer, and the R1 field specifies the first operand, which is 32 or 64 bits in length, depending on the instruction. For the instruction TEST UNDER MASK (TMHH, TMHL, TMLH, TMML), the contents of the I2 field are used as a mask, and the R1 field specifies the first operand, which is 64 bits in length. For the instructions INSERT IMMEDIATE, AND IMMEDIATE, OR IMMEDIATE, and LOAD LOGICAL IMMEDIATE, the contents of the I2 field are used as an unsigned binary integer or a logical value, and the R1 field specifies the first operand, which is 64 bits in length.

For the relative-branch instructions in the RI and RSI formats, the contents of the 16-bit I2 field are used as a signed binary integer designating a number of halfwords. This number, when added to the address of the branch instruction, specifies the branch address. In the RIL format, the I2 field is 32 bits and is used in the same way.

Storage Operands
The use of B and R fields to designate access registers to refer to storage operands is described in “Access-Register-Specified Address Spaces” on page 5-44.

In the RSL, SI, SS, and SSE formats, the contents of the general register designated by the B1 field are added to the contents of the D1 field to form the first-operand address. In the RS, RSY, S, SIY, SS, and SSE formats, the contents of the general register designated by the B2 field are added to the contents of the D2 field or DH2 and DL2 fields to form the second-operand address. In the RX, RXE, RXF, and RXY formats, the contents of the general registers designated by the X2 and B2 fields are added to the contents of the D2 field or DH2 and DL2 fields to form the second-operand address.

When a general register contains a 24-bit or 32-bit length of a storage operand, the length is an unsigned binary integer, except that it is signed for COMPARE UNTIL SUBSTRING EQUAL, with a negative value treated as zero. Similarly, the contents of an L, L1, or L2 field of an instruction are an unsigned binary integer.

In the SS format with a single, eight-bit length field, for the instructions AND (NC), EXCLUSIVE OR (XC), MOVE (MVC), MOVE NUMERICS, MOVE ZONES, and OR (OC), L specifies the number of additional operand bytes to the right of the byte designated by the first-operand address. Therefore, the length in bytes of the first operand is 1-256, corresponding to a length code in L of 0-255. Storage results replace the first operand and are never stored outside the field specified by the address and length. In this format, the second operand has the same length as the first operand. There are variations of the preceding definition that apply to EDIT, EDIT AND MARK, PACK ASCII, PACK UNICODE, TRANSLATE, TRANSLATE AND TEST, UNPACK ASCII, and UNPACK UNICODE.

In the SS format with two length fields, and in the RSL format, L1 specifies the number of additional operand bytes to the right of the byte designated by the first-operand address. Therefore, the length in bytes of the first operand is 1-16, corresponding to a length code in L1 of 0-15. Similarly, L2 specifies the number of additional operand bytes to the right of the location designated by the
second-operand address. Results replace the first operand and are never stored outside the field specified by the address and length. If the first operand is longer than the second, the second operand is extended on the left with zeros up to the length of the first operand. This extension does not modify the second operand in storage.

In the SS format with two R fields, as used by the MOVE TO PRIMARY, MOVE TO SECONDARY, and MOVE WITH KEY instructions, the contents of the general register specified by the R₁ field are a 32-bit unsigned value called the true length. The operands are both of a length called the effective length. The effective length is equal to the true length or 256, whichever is less. The instructions set the condition code to facilitate programming a loop to move the total number of bytes specified by the true length. The SS format with two R fields is also used to specify a range of registers and two storage operands for the LOAD MULTIPLE DISJOINT instruction and to specify one or two registers and one or two storage operands for the PERFORM LOCKED OPERATION instruction.

Address Generation

Trimodal Addressing

Bits 31 and 32 of the current PSW are the addressing-mode bits. Bit 31 is the extended-addressing-mode bit, and bit 32 is the basic-addressing-mode bit. These bits control the size of the effective address produced by address generation. When bits 31 and 32 of the current PSW both are zeros, the CPU is in the 24-bit addressing mode, and 24-bit instruction and operand effective addresses are generated. When bit 31 of the current PSW is zero and bit 32 is one, the CPU is in the 31-bit addressing mode, and 31-bit instruction and operand effective addresses are generated. When bits 31 and 32 of the current PSW are both one, the CPU is in the 64-bit addressing mode, and 64-bit instruction and operand effective addresses are generated.

Execution of instructions by the CPU involves generation of the addresses of instructions and operands. This section describes address generation as it applies to most instructions. In some instructions, the operation performed does not follow the general rules stated in this section. All of these exceptions are explicitly identified in the individual instruction descriptions.

Sequential Instruction-Address Generation

When an instruction is fetched from the location designated by the current PSW, the instruction address is increased by the number of bytes in the instruction, and the instruction is executed. The same steps are then repeated by using the new value of the instruction address to fetch the next instruction in the sequence.

In the 24-bit addressing mode, instruction addresses wrap around, with the halfword at instruction address 2²⁴ - 2 being followed by the halfword at instruction address 0. Thus, in the 24-bit addressing mode, any carry out of PSW bit position 104, as a result of updating the instruction address, is lost. In the 31-bit or 64-bit addressing mode, instruction addresses similarly wrap around, with the halfword at instruction address 2³¹ - 2 or 2⁶⁴ - 2, respectively, followed by the halfword at instruction address 0. A carry out of PSW bit position 97 or 64, respectively, is lost.

Operand-Address Generation

Formation of the Intermediate Value

An operand address that refers to storage is derived from an intermediate value, which either is contained in a register designated by an R field in the instruction or is calculated from the sum of three binary numbers: base address, index, and displacement.

The base address (B) is a 64-bit number contained in a general register specified by the program in a four-bit field, called the B field, in the instruction. Base addresses can be used as a means of independently addressing each program and data area. In array-type calculations, it can designate the location of an array, and, in record-type processing, it can identify the record. The base address provides for addressing the entire storage. The base address may also be used for indexing.

The index (X) is a 64-bit number contained in a general register designated by the program in a four-bit field, called the X field, in the instruction.
It is included only in the address specified by the RX-, RXE-, RXF-, and RXY-format instructions. The RX-, RXE-, RXF-, and RXY-format instructions permit double indexing; that is, the index can be used to provide the address of an element within an array.

The displacement (D) is a 12-bit or 20-bit number contained in a field, called the D field, in the instruction. A 12-bit displacement is unsigned and provides for relative addressing of up to 4,095 bytes beyond the location designated by the base address. A 20-bit displacement is signed and provides for relative addressing of up to 524,287 bytes beyond the base-address location or of up to 524,288 bytes before it. In array-type calculations, the displacement can be used to specify one of many items associated with an element. In the processing of records, the displacement can be used to identify items within a record.

A 12-bit displacement is in bit positions 20-31 of instructions of certain formats (see Figure 5-1 on page 5-3). In instructions of some formats, a second 12-bit displacement also is in the instruction, in bit positions 36-47.

A 20-bit displacement is in instructions of only the RSY, RXY, or SIY format. In these instructions, the D field consists of a DL (low) field in bit positions 20-31 and of a DH (high) field in bit positions 32-39. When the long-displacement facility is installed, the numeric value of the displacement is formed by appending the contents of the DH field on the left of the contents of the DL field. When the long-displacement facility is not installed, the numeric value of the displacement is formed by appending eight zero bits on the left of the contents of the DL field, and the contents of the DH field are ignored.

Formation of the Operand Address
The generated operand address is always 64 bits long, and the bits are numbered 0-63. The manner in which the generated address is obtained from the intermediate value depends on the current addressing mode. In the 24-bit addressing mode, bits 0-39 of the intermediate value are ignored, bits 0-39 of the generated address are forced to be zeros, and bits 40-63 of the intermediate value become bits 40-63 of the generated address. In the 31-bit addressing mode, bits 0-32 of the intermediate value are ignored, bits 0-32 of the generated address are forced to be zero, and bits 33-63 of the intermediate value become bits 33-63 of the generated address. In the 64-bit addressing mode, bits 0-63 of the intermediate value become bits 0-63 of the generated address.

Programming Note: Negative values may be used in index and base-address registers. Bits 0-32 of these values are ignored in the 31-bit addressing mode, and bits 0-39 are ignored in the 24-bit addressing mode.
Branch-Address Generation

Formation of the Intermediate Value
For branch instructions, the address of the next instruction to be executed when the branch is taken is called the branch address. Depending on the branch instruction, the instruction format may be RR, RRE, RX, RXY, RS, RSY, RSI, RI, RIE, or RIL.

In the RS, RSY, RX, and RXY formats, the branch address is specified by a base address, a displacement, and, in the RX and RXY formats, an index. In these formats, the generation of the intermediate value follows the same rules as for the generation of the operand-address intermediate value.

In the RR and RRE formats, the contents of the general register designated by the R field are used as the intermediate value from which the branch address is formed. General register 0 cannot be designated as containing a branch address. A value of zero in the R field causes the instruction to be executed without branching.

The relative-branch instructions are in the RSI, RI, RIE, and RIL formats. In the RSI, RI, and RIE formats for the relative-branch instructions, the contents of the I field are treated as a 16-bit signed binary integer designating a number of halfwords. In the RIL format, the contents of the I field are treated as a 32-bit signed binary integer designating a number of halfwords. The branch address is the number of halfwords designated by the I field added to the address of the relative-branch instruction.

The 64-bit intermediate value for a relative branch instruction in the RSI, RI, RIE, or RIL format is the sum of two addends, with overflow from bit position 0 ignored. In the RSI, RI, or RIE format, the first addend is the contents of the I field with one zero bit appended on the right and 47 bits equal to the sign bit of the contents appended on the left. In the RIL format, the first addend is the contents of the I field with one zero bit appended on the right and 31 bits equal to the sign bit of the contents appended on the left. In all formats, the second addend is the 64-bit address of the branch instruction. The address of the branch instruction is the instruction address in the PSW before that address is updated to address the next sequential instruction, or it is the address of the target of the EXECUTE instruction if EXECUTE is used. If EXECUTE is used in the 24-bit or 31-bit addressing mode, the address of the branch instruction is the target address with 40 or 33 zeros, respectively, appended on the left.

Formation of the Branch Address
The branch address is always 64 bits long, with the bits numbered 0-63. The branch address replaces bits 64-127 of the current PSW.

The manner in which the branch address is obtained from the intermediate value depends on the addressing mode. For those branch instructions which change the addressing mode, the new addressing mode is used. In the 24-bit addressing mode, bits 0-39 of the intermediate value are ignored, bits 0-39 of the branch address are made zeros, and bits 40-63 of the intermediate value become bits 40-63 of the branch address. In the 31-bit addressing mode, bits 0-32 of the intermediate value are ignored, bits 0-32 of the branch address are made zeros, and bits 33-63 of the intermediate value become bits 33-63 of the branch address. In the 64-bit addressing mode, bits 0-63 of the intermediate value become bits 0-63 of the branch address.

For several branch instructions, branching depends on satisfying a specified condition. When the condition is not satisfied, the branch is not taken, normal sequential instruction execution continues, and the branch address is not used. When a branch is taken, bits 0-63 of the branch address replace bits 64-127 of the current PSW. The branch address is not used to access storage as part of the branch operation.

A specification exception due to an odd branch address and access exceptions due to fetching of the instruction at the branch location are not recognized as part of the branch operation but instead are recognized as exceptions associated with the execution of the instruction at the branch location.

A branch instruction, such as BRANCH AND SAVE, can designate the same general register for branch-address computation and as the location of an operand. Branch-address computation is completed before the remainder of the operation is performed.
Instruction Execution and Sequencing

The program-status word (PSW), described in Chapter 4, “Control,” contains information required for proper program execution. The PSW is used to control instruction sequencing and to hold and indicate the status of the CPU in relation to the program currently being executed. The active or controlling PSW is called the current PSW.

Branch instructions perform the functions of decision making, loop control, and subroutine linkage. A branch instruction affects instruction sequencing by introducing a new instruction address into the current PSW. The relative-branch instructions with a 16-bit I\_2 field allow branching to a location at an offset of up to plus 64K - 2 bytes or minus 64K bytes relative to the location of the branch instruction, without the use of a base register. The relative-branch instructions with a 32-bit I\_2 field allow branching to a location at an offset of up to plus 4G - 2 bytes or minus 4G bytes relative to the location of the branch instruction, without the use of a base register.

Decision Making

Facilities for decision making are provided by the BRANCH ON CONDITION, BRANCH RELATIVE ON CONDITION, and BRANCH RELATIVE ON CONDITION LONG instructions. These instructions inspect a condition code that reflects the result of a majority of the arithmetic, logical, and I/O operations. The condition code, which consists of two bits, provides for four possible condition-code settings: 0, 1, 2, and 3.

The specific meaning of any setting depends on the operation that sets the condition code. For example, the condition code reflects such conditions as zero, nonzero, first operand high, equal, overflow, and subchannel busy. Once set, the condition code remains unchanged until modified by an instruction that causes a different condition code to be set. See Appendix C, “Condition-Code Settings” on page C-1 for a summary of the instructions which set the condition code.

Loop Control

Loop control can be performed by the use of BRANCH ON CONDITION, BRANCH RELATIVE ON CONDITION, and BRANCH RELATIVE ON CONDITION LONG to test the outcome of address arithmetic and counting operations. For some particularly frequent combinations of arithmetic and tests, BRANCH ON COUNT, BRANCH ON INDEX HIGH, and BRANCH ON INDEX LOW OR EQUAL are provided, and relative-branch equivalents of these instructions are also provided. These branches, being specialized, provide increased performance for these tasks.

Subroutine Linkage without the Linkage Stack

This section describes only the methods for subroutine linkage that do not use the linkage stack. For the linkage extensions provided by the linkage stack, see “Linkage-Stack Introduction” on page 5-69 (Those extensions include a different method of operation of the PROGRAM CALL instruction and also the BRANCH AND STACK and PROGRAM RETURN instructions.)

Simple Branch Instructions

Subroutine linkage when a change of the addressing mode is not required is provided by the BRANCH AND LINK and BRANCH AND SAVE instructions. (This discussion of BRANCH AND SAVE applies also to BRANCH RELATIVE AND SAVE and BRANCH RELATIVE AND SAVE LONG.) Both of these instructions permit not only the introduction of a new instruction address but also the preservation of a return address and associated information. The return address is the address of the instruction following the branch instruction in storage, except that it is the address of the instruction following an EXECUTE instruction that has the branch instruction as its target.

Both BRANCH AND LINK and BRANCH AND SAVE have an R\_1 field. They form a branch address by means of fields that depend on the instruction. The operations of the instructions are summarized as follows:

- In the 24-bit addressing mode, both instructions place the return address in bit positions 40-63 of general register R\_1 and leave bits 0-31 of that register unchanged.
BRANCH AND LINK places the instruction-length code for the instruction and also the condition code and program mask from the current PSW in bit positions 32-39 of general register R₁. BRANCH AND SAVE places zeros in those bit positions.

- In the 31-bit addressing mode, both instructions place the return address in bit positions 33-63 and a one in bit position 32 of general register R₁, and they leave bits 0-31 of the register unchanged.
- In the 64-bit addressing mode, both instructions place the return address in bit positions 0-63 of general register R₁.
- In any addressing mode, both instructions generate the branch address under the control of the current addressing mode. The instructions place bits 0-63 of the branch address in bit positions 64-127 of the PSW. In the RR format, both instructions do not perform branching if the R₂ field of the instruction is zero.

It can be seen that, in the 24-bit or 31-bit addressing mode, BRANCH AND SAVE places the basic-addressing-mode bit, bit 32 of the PSW, in bit position 32 of general register R₁. BRANCH AND LINK does so in the 31-bit addressing mode.

The instructions BRANCH AND SAVE AND SET MODE and BRANCH AND SET MODE are for use when a change of the addressing mode is required during linkage. These instructions have R₁ and R₂ fields. The operations of the instructions are summarized as follows:

- BRANCH AND SAVE AND SET MODE sets the contents of general register R₁ the same as BRANCH AND SAVE. In addition, the instruction places the extended-addressing-mode bit, bit 31 of the PSW, in bit position 63 of the register.
- BRANCH AND SET MODE, if R₁ is nonzero, performs as follows. In the 24- or 31-bit mode, it places bit 32 of the PSW in bit position 32 of general register R₁, and it leaves bits 0-31 and 33-63 of the register unchanged. Note that bit 63 of the register should be zero if the register contains an instruction address. In the 64-bit mode, the instruction places bit 31 of the PSW (a one) in bit position 63 of general register R₁, and it leaves bits 0-62 of the register unchanged.
- When R₂ is nonzero, both instructions set the addressing mode and perform branching as follows. Bit 63 of general register R₂ is placed in bit position 31 of the PSW. If bit 63 is zero, bit 32 of the register is placed in bit position 32 of the PSW. If bit 63 is one, PSW bit 32 is set to one. Then the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new addressing mode. The instructions place bits 0-63 of the branch address in bit positions 64-127 of the PSW. Bit 63 of general register R₂ remains unchanged and, therefore, may be one upon entry to the called program. If R₂ is the same as R₁, the results in the designated general register are as specified for the R₁ register.

The operations of the simple branch instructions are summarized in Figure 5-2 on page 5-13. For contrast, the figure also shows the BRANCH ON COUNT instruction, which is not for use in linkage, and the LOAD ADDRESS and LOAD ADDRESS EXTENDED instructions.

Programming Notes:

1. A called program that is entered in the 64-bit addressing mode can use bit 63 of the entry-point register to determine the instruction used to perform the call and, thus, the instruction that must be used to perform the return linkage. If bit 63 is zero, BRANCH AND SAVE (BAS or BASR) (or possibly BAL, BALR, BRAS, or BRASL) was used, the addressing mode of the caller is not indicated in the return register, and BRANCH ON CONDITION (BCR) must be used to return without changing the addressing mode during the return. If bit 63 is the entry-point register is one, BASSM or BSM was used, the addressing mode of the caller is indicated in the return register (or at least can be, in the case of BSM), and BSM must be used to return and restore the addressing mode of the caller.

2. When BSM is executed in the 24-bit or 31-bit addressing mode and used in a forward linkage to set the 64-bit mode, and the R₁ and R₂ of the instruction are the same value, bit 63 of the designated general register does not, upon entry to the called program, correctly indicate the mode of the calling program. (The bit is one, instead of zero,
because the program set bit 63 of the R$_2$ register to one and the instruction does not change bit 63 of the R$_1$ register in the 24- or 31-bit mode.) BASSM always correctly indicates the addressing mode of the calling program.

3. If an entry point can be branched to in the 64-bit addressing mode either by BAS or BASR or by BASSM or BSM, one must be subtracted from the entry-point register in the BASSM or BSM case if the register is to be named in a USING statement that provides addressability.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Mode</th>
<th>Address Placed in GR R$_i$</th>
<th>Branch or 2nd-Op Adr.</th>
<th>R$_2$ Bit 63</th>
<th>PSW Bit 31</th>
<th>PSW Bit 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>BALR*/BAL</td>
<td>RR/RX</td>
<td>24</td>
<td>U *** *** IA</td>
<td>SIA SIA LSExc U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
<td>U BAM IA IA</td>
<td>SIA SIA LSExc U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>IA IA IA IA</td>
<td>SIA SIA LSExc U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASR*/BAS/BRAS/BRASL</td>
<td>RR/RX/RI/RIL</td>
<td>24/31</td>
<td>U BAM IA IA</td>
<td>SIA SIA LSExc U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>IA IA IA IA</td>
<td>SIA SIA LSExc U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASSM*</td>
<td>RR</td>
<td>24/31</td>
<td>U BAM IA IA</td>
<td>SIA SIA 0 0 R232</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>24/31</td>
<td>U BAM IA IA</td>
<td>SIA SIA 1G 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>IA IA IA 1</td>
<td>SIA SIA 0 0 R232</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>IA IA IA 1</td>
<td>SIA SIA 1G 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSM**</td>
<td>RR</td>
<td>24/31</td>
<td>U BAM U U U</td>
<td>SIA SIA 0 0 R232</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>24/31</td>
<td>U BAM U U U</td>
<td>SIA SIA 1G 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>U U U 1</td>
<td>SIA SIA 0 0 R232</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>U U U 1</td>
<td>SIA SIA 1G 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCTR*/BCT/BCTGR*/BCTG</td>
<td>RR/RX/RRE/RXY</td>
<td>24/31</td>
<td>NLA NLA NLA NLA</td>
<td>SIA SIA LSExc U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>NLA NLA NLA NLA</td>
<td>SIA SIA LSExc U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA/LAE</td>
<td>RX/RX</td>
<td>24/31</td>
<td>U 0 Op2Ad Op2Ad Op2Ad</td>
<td>FZ SR1 0/1 U U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>Op2Ad Op2Ad Op2Ad SR1</td>
<td>SR1 0/1 U U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-2 (Part 1 of 2). Summary of Simple Branch Linkage Instructions and Other Instructions
Explanation:

- The address does not exist, or the bit has no special effect.

* The action associated with the R2 field is not performed if the field is zero.

** The action associated with the R1 or R2 field is not performed if the field is zero.

*** The instruction-length code, condition code, and program mask are saved in bit positions 32-39 of the link address, and bits 40-63 of the updated instruction address are saved in bit positions 40-63.

0/1 Bit 63 can be zero or one.

1G0 Bit 63 is one and is left one, but the branch address is generated as if the bit is zero.

BAM Bit 32 of the link address is set with the basic-addressing-mode bit, bit 32 of the PSW.

FZ Bits 0-32 of the second-operand address are forced to zeros in the 24-bit or 31-bit addressing mode.

IA Bits of the link address are set with the updated instruction address as shown.

LSExc A late specification exception is recognized if the bit is one.

NLA The instruction does not produce a link address. (The instruction is shown simply as an example of a non-linkage branch instruction.)

Op2Ad Bits of the address in general register R1 are set with the corresponding bits of the second-operand address as shown.

R232 The basic-addressing-mode bit, bit 32 of the PSW, is set with bit 32 of general register R2.

SIA Bits 0-63 of the branch address are used to set the instruction address in the PSW. Bits 0-39 of the branch address are forced to zeros in the 24-bit addressing mode. Bits 0-32 are forced to zeros in the 31-bit addressing mode.

SR1 Bits of the second-operand address are used to set the corresponding bits of the address in the R1 general register as shown. Bits 0-39 of the second-operand address are forced to zeros in the 24-bit addressing mode. Bits 0-32 are forced to zeros in the 31-bit addressing mode.

U Unchanged.

Figure 5-2 (Part 2 of 2). Summary of Simple Branch Linkage Instructions and Other Instructions
Other Linkage Instructions

Note: The discussion in this section of PROGRAM TRANSFER applies equally to PROGRAM TRANSFER WITH INSTANCE, which differs from PROGRAM TRANSFER only as described in "ASN-Second-Table-Entry Instance Number and ASN Reuse" on page 3-19. PROGRAM TRANSFER WITH INSTANCE is not further referred to in this section.

Linkage between a problem-state program and the supervisor or monitoring program is provided by means of the SUPERVISOR CALL and MONITOR CALL instructions.

The instructions PROGRAM CALL and PROGRAM TRANSFER provide the facility for linkage between programs of different authority and in different address spaces. PROGRAM CALL permits linkage to a number of preassigned programs that may be in either the problem or the supervisor state and may be in either the same address space or an address space different from that of the caller. It permits a change between the 24-bit and 31-bit addressing modes, and it permits an increase of PSW-key-mask authority, which authorizes the execution of the SET PSW KEY FROM ADDRESS instruction and also other functions. In general, PROGRAM CALL is used to transfer control to a program of higher authority. PROGRAM TRANSFER permits a change of the instruction address and address space and a change between the 24-bit and 31-bit addressing modes. PROGRAM TRANSFER also permits a reduction of PSW-key-mask authority and a change from the supervisor to the problem state. In general, it is used to transfer control from one program to another of equal or lower authority.

When a calling linkage is to increase authority, the calling linkage can be performed by PROGRAM CALL and the return linkage by PROGRAM TRANSFER. Alternatively, when the calling linkage is to decrease authority, the calling linkage can be performed by PROGRAM TRANSFER and the return linkage by PROGRAM CALL.

The operation of PROGRAM CALL is controlled by means of an entry-table entry, which is located as part of a table-lookup process during the execution of the instruction. The entry-table entry specifies either a basic (nonstacking) operation or the stacking operation described in "Linkage-Stack Introduction" on page 5-69. The instruction causes the primary address space to be changed only when the ASN in the entry-table entry is nonzero. When the primary address space is changed, the operation is called PROGRAM CALL with space switching (PC-ss). When the primary address space is not changed, the operation is called PROGRAM CALL to current primary (PC-cp).

PROGRAM TRANSFER specifies the address space which is to become the new primary address space. When the primary address space is changed, the operation is called PROGRAM TRANSFER with space switching (PT-ss). When the primary address space is not changed, the operation is called PROGRAM TRANSFER to current primary (PT-cp).

Basic PROGRAM CALL, and PROGRAM TRANSFER, can be executed successfully in either a basic (24-bit or 31-bit) addressing mode or the extended (64-bit) addressing mode. They do not provide a change between a basic addressing mode and the extended addressing mode.

The BRANCH AND SET AUTHORITY instruction can improve performance by replacing a PT-cp instruction used to perform a calling linkage in which PSW-key-mask authority is reduced, and by replacing a PC-cp instruction used to perform the associated return linkage in which PSW-key-mask authority is restored. BRANCH AND SET AUTHORITY also permits changes between the supervisor and problem states, and it can replace SET PSW KEY FROM ADDRESS by changing the PSW key during the linkage. The calling-linkage operation is called BRANCH AND SET AUTHORITY in the base-authority state (BSA-ba), and the return-linkage operation is called BRANCH AND SET AUTHORITY in the reduced-authority state (BSA-ra).

The BRANCH IN SUBSPACE GROUP instruction allows linkage within a group of address spaces called a subspace group, where one address space in the group is called the base space and the others are called subspaces. It is intended that each subspace contain a different subset of the storage in the base space, that the base space and each subspace contain a subsystem control program, such as CICS, and application programs, and that each subspace contain the...
data for a single transaction being processed under the subsystem control program. The placement of the data for each transaction in a different subspace prevents a program that is being executed to process one particular transaction from erroneously damaging the data of other transactions. It is intended that the primary address space be the base space when the control program is being executed, and that it be the subspace for a transaction when an application program is being executed to process that transaction. BRANCH IN SUBSPACE GROUP changes not only the instruction address in the PSW but also the primary address-space-control element in control register 1. BRANCH IN SUBSPACE GROUP does not change the primary ASN in control register 4 or the primary-ASN-second-table-entry origin in control register 5, and, therefore, the base space and the subspaces all are associated with the same ASN, and the programs in those address spaces all are of equal authority.

Although a subspace is intended to be a subset of the base space as described above, BRANCH IN SUBSPACE GROUP does not require this, and the instruction may be useful in ways other than as described above.

BRANCH IN SUBSPACE GROUP uses an access-list-entry token (ALET) in an access register as an identifier of the address space that is to receive control. The instruction saves the updated instruction address to permit a return linkage, but it does not save an identifier of the address space from which control was transferred. However, an ALET equal to 00000000 hex, called ALET 0, can be used to return from a subspace to the base space, and an ALET equal to 00000001 hex, called ALET 1, can be used to return from the base space to the subspace that last had control.

The SET ADDRESSING MODE (SAM24, SAM31, SAM64) instruction can assist in linkage by setting the 24-bit, 31-bit, or 64-bit addressing mode either before or after a linkage operation.

The RESUME PROGRAM instruction is intended for use by a problem-state interruption-handling program to return to the interrupted program. The interruption-handling program can use LOAD ACCESS MULTIPLE and LOAD MULTIPLE instructions to restore the contents of the interrupted program's access and general registers from a save area, except for the contents of one access-and-general register pair. The interruption-handling program then can use RESUME PROGRAM to restore the contents of certain PSW fields, including the instruction address, and also the contents of the remaining access-and-general pair from the save area, with that pair first being used by RESUME PROGRAM to address the save area.

The TRAP instruction (TRAP2, TRAP4) can overlay instructions in an application program and give control to a trap program for performing fix-ups of data used by the application program. The RESUME PROGRAM instruction can be used to return control from the trap program to the application program.

The linkage instructions provided and the functions performed by each are summarized in Figure 5-3 on page 5-17.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Save</th>
<th>Set</th>
<th>Save</th>
<th>Set</th>
<th>Save</th>
<th>Set</th>
<th>Save</th>
<th>Set</th>
<th>Save</th>
<th>Set</th>
<th>Save</th>
<th>Set</th>
<th>PSW-Key Mask Changed in CR3</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>BALR</td>
<td>RR</td>
<td>Yes*</td>
<td>R2</td>
<td>BAM31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R2^1</td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>RX</td>
<td>Yes*</td>
<td>Yes</td>
<td>BAM31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASR</td>
<td>RR</td>
<td>Yes</td>
<td>R2</td>
<td>BAM</td>
<td>R2</td>
<td>Yes</td>
<td>R2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R2^1</td>
<td></td>
</tr>
<tr>
<td>BAS</td>
<td>RX</td>
<td>Yes</td>
<td>Yes</td>
<td>BAM</td>
<td>R2</td>
<td>Yes</td>
<td>R2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BASM</td>
<td>RR</td>
<td>Yes</td>
<td>R2</td>
<td>BAM</td>
<td>R2</td>
<td>Yes</td>
<td>R2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRAS</td>
<td>RI</td>
<td>Yes</td>
<td>Yes</td>
<td>BAM</td>
<td>R2</td>
<td>Yes</td>
<td>R2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRASL</td>
<td>RIL</td>
<td>Yes</td>
<td>Yes</td>
<td>BAM</td>
<td>R2</td>
<td>Yes</td>
<td>R2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSA-ba</td>
<td>RRE</td>
<td>Yes</td>
<td>Yes</td>
<td>BAM</td>
<td>BAM</td>
<td>-</td>
<td>Yes</td>
<td>Yes^4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&quot;AND&quot; R1^5</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSA-ra</td>
<td>RRE</td>
<td>R1</td>
<td>Yes</td>
<td>R1</td>
<td>BAM</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSG</td>
<td>RRE</td>
<td>Yes</td>
<td>Yes</td>
<td>R1</td>
<td>BAM</td>
<td>-</td>
<td>Yes</td>
<td>R1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSM</td>
<td>RR</td>
<td>R2</td>
<td>R1</td>
<td>R1</td>
<td>BAM</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>EAM64</td>
<td>R2</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MC^2</td>
<td>SI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC-cp</td>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>BAM</td>
<td>BAM</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&quot;OR&quot; EKM</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC-ss</td>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>BAM</td>
<td>BAM</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>&quot;OR&quot; EKM</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT-cp or PTI-cp</td>
<td>RRE</td>
<td>R2</td>
<td>R2</td>
<td>BAM</td>
<td>R2</td>
<td>-</td>
<td>R2</td>
<td>-</td>
<td>&quot;AND&quot; R1</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT-ss or PTI-ss</td>
<td>RRE</td>
<td>R2</td>
<td>R2</td>
<td>BAM</td>
<td>R2</td>
<td>-</td>
<td>R2</td>
<td>-</td>
<td>&quot;AND&quot; R1</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RP</td>
<td>S</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>&quot;AND&quot; R1</td>
<td>Yes</td>
</tr>
<tr>
<td>SAM24</td>
<td>E</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>0</td>
<td>Yes</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAM31</td>
<td>E</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAM64</td>
<td>E</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVC^2</td>
<td>RR</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP2</td>
<td>E</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP4</td>
<td>S</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

- No

* In the 24-bit addressing mode, the instruction-length code, condition code, and program mask are saved in bit positions 32-39 of the R1 general register.

Figure 5-3 (Part 1 of 2). Summary of Linkage Instructions without the Linkage Stack
**Explanation (Continued):**

** A change from the supervisor to the problem state is allowed; a privileged-operation exception is recognized when a change from the problem to the supervisor state is specified.

# Monitor-mask bits provide a means of disallowing linkage, or enabling linkage, for selected classes of events.

1 The action takes place only if the associated R field in the instruction is nonzero.

2 MC and SVC, as part of the interruption, save the entire current PSW and load a new PSW.

3 The primary address-space-control element is set even though the PASN is not set.

4 The problem state is set.

5 The PSW key also is set from general register R1.

BAM The basic-addressing-mode bit is saved or set only in the 24-bit or 31-bit addressing mode.

BAM31 The basic-addressing-mode bit is saved only in the 31-bit addressing mode.

EAM64 The extended-addressing-mode bit is saved only in the 64-bit addressing mode.

R1 The field or bit is saved in general register R1.

R2 The field or bit is set from general register R2.

---

**Programming Note:** This note describes the simple branch-type linkage instructions that were included in 370-XA and carried forward to ESA/370, ESA/390, and z/Architecture. To give the reader a better understanding of the utility and intended usage of these linkage instructions, the following paragraphs in this note describe various program linkages and conventions and the use of the linkage instructions in these situations.

The linkage instructions were originally provided to permit System/370 programs to operate with no modification or only slight modification on 370/XA (and successor) systems and also to provide additional function for those programs which were designed to take advantage of the 31-bit addressing of 370/XA. The instructions provided the capability for both old and new programs to coexist in storage and to communicate with each other. The instructions now have been enhanced to permit usage of the 64-bit addressing of z/Architecture.

With respect to System/370 programs, it is assumed that old, unmodified programs operate in the 24-bit addressing mode and call, or directly communicate with, other programs operating in the 24-bit addressing mode only. Modified programs normally operate in the 24-bit addressing mode but may have called programs which operate in either the 24-bit or 31-bit addressing mode. They and also modified 370-XA, ESA/370, and ESA/390 programs now may call programs that operate in the 24-bit, 31-bit, or 64-bit addressing mode. New programs may be written to operate in any addressing mode, and, in some cases, a program may be written such that it can be invoked in any addressing mode.

BRANCH AND SAVE AND SET MODE (BASSM) is intended to be the principal calling instruction to subroutines outside of an assembler/linkage-editor control section (CSECT), for use by all new programs and particularly by programs that must change the addressing mode during the linkage. The calling sequence has normally been:

```
L 15,ACON
BASSM 14,15
...
EXTRN SUB
ACON DC A(X'80000000'+SUB)
```

where ACON is an A-type address constant, and the X'80000000' should be present to give control in the 31-bit addressing mode or should be omitted to give control in the 24-bit addressing mode.

The return from such a routine normally is:
BSM  0,14

It is assumed that the A-type address constant will be extended so it may be an eight-byte field containing a 64-bit entry-point address, with bit 63 of the address indicating, when one, that the entry is in the 64-bit addressing mode. This extended constant is shown here as “ACONE.” The calling sequence would normally be:

```
LG 15,ACONE
BASSM 14,15
```

`EXTRN SUB
ACONE DC AD(X’1'+SUB)`

The return from such a routine would normally be:

```
BSM 0,14
```

When a change of the addressing mode is not required, BRANCH AND LINK or BRANCH AND SAVE should be used instead of BASSM.

The BRANCH AND LINK (BAL, BALR) instruction is provided primarily for compatibility with System/370. It is defined to operate in the 31-bit and 64-bit addressing modes to increase the probability that an old, straightforward program can be modified to operate in those addressing modes with minimal or no change. It is recommended, however, that BRANCH AND SAVE (BAS and BASR) be used instead and that BRANCH AND LINK be avoided since it places nonzero information in bit positions 32-39 of the general register in the 24-bit addressing mode, which may lead to problems and may decrease performance. BRANCH RELATIVE AND SAVE and BRANCH RELATIVE AND SAVE LONG may be used instead of BRANCH AND SAVE.

It is assumed that the normal return from a subroutine called in the 24-bit or 31-bit addressing mode by BRANCH AND LINK (BAL or BALR) will be:

```
BCR 15,14
```

However, the standard “return instruction”:

```
BSM 0,14
```

operates correctly for all cases except for a calling BAL executed in the 24-bit addressing mode. In the 24-bit addressing mode, BAL causes an ILC of 10 to be placed in bit positions 32 and 33 of the link register. Thus, a BSM would return in the 31-bit addressing mode. Note that an EXECUTE of BALR in the 24-bit addressing mode also causes the same ILC effect.

The BRANCH AND SAVE (BAS, BASR) instruction is provided to be used for subroutine linkage to any program either within the same CSECT or known to be in the same addressing mode. BASR with the R2 field 0 is also useful for obtaining addressability to the instruction stream by getting a 31-bit address, uncluttered by leftmost fields, in the 24-bit addressing mode.

The instruction for returning from a routine called in the 24-bit or 31-bit addressing mode by BRANCH AND SAVE (BAS or BASR) may be either:

```
BCR 15,14
```

or:

```
BSM 0,14
```

The instruction for returning from a routine called in the 64-bit addressing mode by BAS or BASR must be BCR; BSM would set the 24-bit or 31-bit addressing mode, depending on bit 32 of the link register (an address bit), because bit 63 of the link register (the rightmost bit of an instruction address) is zero. BSM can always be used as the return instruction if BASSM is used as the calling instruction.

In some cases, it may be desirable to rewrite a program that is called by an old program which has not been rewritten. In such a case, the old program, which operates in the 24-bit or 31-bit addressing mode, will be given the address of an intermediate program that will set up the correct entry and return modes and then call the rewritten program. Such an intermediate program is sometimes referred to as a glue module. The instruction BRANCH AND SET MODE (BSM) with a nonzero R1 field provides the function necessary to perform this operation efficiently. This is shown in Figure 5-4 on page 5-20 for a linkage from a 24-bit-mode program to a 31-bit-mode program.

Note that the “BSM 14,15” in the glue module causes either an indication of the 64-bit addressing mode to be saved in bit position 63 of general register 14 or an indication of one of the 24-bit and 31-bit addressing modes to be saved in bit position 32 of the register, and that the other bits of the register are unchanged. Thus, when “BSM 0,14” is executed in the new program, control passes directly back to the old program without passing through the glue module again.
The glue module could give control to a program in the 64-bit addressing mode and possibly above the 2G-byte boundary by loading an eight-byte A-type address constant, with bit 63 set to one, instead of a four-byte A-type address constant.

**Interruptions**

Interruptions permit the CPU to change state as a result of conditions external to the system, in subchannels or input/output (I/O) devices, in other CPUs, or in the CPU itself. Details are to be found in Chapter 6, “Interruptions.”

Six classes of interruption conditions are provided: external, I/O, machine check, program, restart, and supervisor call. Each class has two related PSWs, called old and new, in permanently assigned real storage locations. In all classes, an interruption involves storing information identifying the cause of the interruption, storing the current PSW at the old-PSW location, and fetching the PSW at the new-PSW location, which becomes the current PSW.

The old PSW contains CPU-status information necessary for resumption of the interrupted program. At the conclusion of the program invoked by the interruption, the instruction LOAD PSW EXTENDED may be used to restore the current PSW to the value of the old PSW.

**Types of Instruction Ending**

Instruction execution ends in one of five ways: completion, nullification, suppression, termination, and partial completion.

Partial completion of instruction execution occurs only for interruptible instructions; it is described in “Interruptible Instructions” on page 5-21.

**Completion**

Completion of instruction execution provides results as called for in the definition of the instruction. When an interruption occurs after the completion of the execution of an instruction, the instruction address in the old PSW designates the next sequential instruction.

**Suppression**

Suppression of instruction execution causes the instruction to be executed as if it specified “no operation.” The contents of any result fields, including the condition code, are not changed. The instruction address in the old PSW on an interruption after suppression designates the next sequential instruction.
Nullification
Nullification of instruction execution has the same effect as suppression, except that when an interruption occurs after the execution of an instruction has been nullified, the instruction address in the old PSW designates the instruction whose execution was nullified (or an EXECUTE instruction, as appropriate) instead of the next sequential instruction.

Termination
Termination of instruction execution causes the contents of any fields due to be changed by the instruction to be unpredictable. The operation may replace all, part, or none of the contents of the designated result fields and may change the condition code if such change is called for by the instruction. Unless the interruption is caused by a machine-check condition, the validity of the instruction address in the PSW, the interruption code, and the ILC are not affected, and the state or the operation of the machine is not affected in any other way. The instruction address in the old PSW on an interruption after termination designates the next sequential instruction.

Programming Note: Although the execution of an instruction is treated as a no-operation when suppression or nullification occurs, stores may be performed as the result of the implicit tracing action associated with some instructions. See “Tracing” on page 4-10.

Interruptible Instructions

Point of Interruption
For most instructions, the entire execution of an instruction is one operation. An interruption is permitted between operations; that is, an interruption can occur after the performance of one operation and before the start of a subsequent operation.

For the following instructions, referred to as interruptible instructions, an interruption is permitted also after partial completion of the instruction:

- COMPARE AND FORM CODEWORD
- COMPARE LOGICAL LONG
- COMPARE UNTIL SUBSTRING EQUAL
- COMPRESSION CALL
- MOVE LONG
- TEST BLOCK
- UPDATE TREE

Unit of Operation
Whenever points of interruption that include those occurring within the execution of an interruptible instruction are discussed, the term “unit of operation” is used. For a noninterruptible instruction, the entire execution consists, in effect, in the execution of one unit of operation.

The execution of an interruptible instruction is considered to consist in the execution of a number of units of operation, and an interruption is permitted between units of operation. The amount of data processed in a unit of operation depends on the particular instruction and may depend on the model and on the particular condition that causes the execution of the instruction to be interrupted.

When an instruction execution consists of a number of units of operation and an interruption occurs after some, but not all, units of operation have been completed, the instruction is said to be partially completed. In this case, the type of ending (completion, nullification, or suppression) is associated with the unit of operation. In the case of termination, the entire instruction is terminated, not just the unit of operation.

An exception may exist that causes the first unit of operation of an interruptible instruction not to be completed. In this case when the ending is nullification or suppression, all operand parameters and result locations remain unchanged, except that the condition code is unpredictable if the instruction is defined to set the condition code.

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored before the event is indicated by an interruption.

Execution of Interruptible Instructions
The execution of an interruptible instruction is completed when all units of operation associated with that instruction are completed. When an interruption occurs after completion, nullification, or suppression of a unit of operation, all preceding units of operation have been completed, and subsequent units of operation and instructions have not been started. The main difference between these types of ending is the handling of the current unit of operation and whether the instruction address stored in the old PSW identifies the current instruction or the next sequential instruction.
At the time of an interruption, changes to storage locations or register contents which are due to be made by instructions following the interrupted instruction have not yet been made.

**Completion:** On completion of the last unit of operation of an interruptible instruction, the instruction address in the old PSW designates the next sequential instruction. The result location for the current unit of operation has been updated. It depends on the particular instruction how the operand parameters are adjusted. On completion of a unit of operation other than the last one, the instruction address in the old PSW designates the interrupted instruction or an EXECUTE instruction, as appropriate. The result location for the current unit of operation has been updated. The operand parameters are adjusted such that the execution of the interrupted instruction is resumed from the point of interruption when the old PSW stored during the interruption is made the current PSW.

**Nullification:** When a unit of operation is nullified, the instruction address in the old PSW designates the interrupted instruction or an EXECUTE instruction, as appropriate. The result location for the current unit of operation remains unchanged. The operand parameters are adjusted such that, if the instruction is reexecuted, execution of the interrupted instruction is resumed with the current unit of operation.

**Suppression:** When a unit of operation is suppressed, the instruction address in the old PSW designates the next sequential instruction. The operand parameters, however, are adjusted so as to indicate the extent to which instruction execution has been completed. If the instruction is reexecuted after the conditions causing the suppression have been removed, the execution is resumed with the current unit of operation.

**Termination:** When an exception which causes termination occurs as part of a unit of operation of an interruptible instruction, the entire operation is terminated, and the contents, in general, of any fields due to be changed by the instruction are unpredictable. On such an interruption, the instruction address in the old PSW designates the next sequential instruction.

The differences among the four types of ending for a unit of operation are summarized in Figure 5-5.

<table>
<thead>
<tr>
<th>Unit of Operation Is</th>
<th>Instruction Address</th>
<th>Operand Parameters</th>
<th>Current Result Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completed Last unit of operation</td>
<td>Next instruction</td>
<td>Depends on the instruction</td>
<td>Changed</td>
</tr>
<tr>
<td>Completed Any other unit of operation</td>
<td>Current instruction</td>
<td>Next unit of operation</td>
<td>Changed</td>
</tr>
<tr>
<td>Nullified</td>
<td>Current instruction</td>
<td>Current unit of operation</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Suppressed</td>
<td>Next instruction</td>
<td>Current unit of operation</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Terminated</td>
<td>Next instruction</td>
<td>Unpredictable</td>
<td>Unpredictable</td>
</tr>
</tbody>
</table>

**Figure 5-5. Types of Ending for a Unit of Operation**

If an instruction is defined to set the condition code, the execution of the instruction makes the condition code unpredictable except when the last unit of operation has been completed.

**Condition-Code Alternative to Interruptibility**

The following instructions are not interruptible instructions but instead may be completed after performing a CPU-determined subportion of the processing specified by the parameters of the instructions:

- CHECKSUM
- COMPARE LOGICAL LONG EXTENDED
- COMPARE LOGICAL LONG UNICODE
- COMPARE LOGICAL STRING
- CONVERT UTF-16 TO UTF-32
- CONVERT UTF-16 TO UTF-8
- CONVERT UTF-32 TO UTF-16
- CONVERT UTF-32 TO UTF-8
- CONVERT UTF-8 TO UTF-16
- CONVERT UTF-8 TO UTF-32
- MOVE LONG EXTENDED
- MOVE LONG UNICODE
- MOVE STRING
- SEARCH STRING
- SEARCH STRING UNICODE
- TRANSLATE EXTENDED
- TRANSLATE ONE TO ONE
- TRANSLATE ONE TO TWO
- TRANSLATE TWO TO ONE
- TRANSLATE TWO TO TWO

When any of the above instructions is completed after performing only a CPU-determined amount of processing instead of all specified processing, the instruction sets condition code 3. On such com-
pletion, the instruction address in the PSW designates the next sequential instruction, and the operand parameters of the instruction have been adjusted so that the processing of the instruction can be resumed simply by branching back to the instruction to execute it again. When the instruction has performed all specified processing, it sets a condition code other than 3.

The points at which any of the above instructions may set condition code 3 are comparable to the points of interruption of an interruptible instruction, and the amount of processing between adjacent points is comparable to a unit of operation of an interruptible instruction. However, since the instruction is not interruptible, each execution is considered the execution of one unit of operation.

Completion with the setting of condition code 3 permits interruptions to occur. Depending on the model and the instruction, condition code 3 may or may not be set when there is not a need for an interruption.

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored before the event is indicated by an interruption.

The COMPARE UNTIL SUBSTRING EQUAL and COMPRESSION CALL instructions both are interruptible instructions and ones that may set condition code 3 after performing a CPU-determined amount of processing.

Programming Notes:

1. Any interruption, other than supervisor call and some program interruptions, can occur after a partial execution of an interruptible instruction. In particular, interruptions for external, I/O, machine-check, restart, and program interruptions for access exceptions and PER events can occur between units of operation.

2. The amount of data processed in a unit of operation of an interruptible instruction depends on the model and may depend on the type of condition which causes the execution of the instruction to be interrupted or stopped. Thus, when an interruption occurs at the end of the current unit of operation, the length of the unit of operation may be different for different types of interruptions. Also, when the stop function is requested during the execution of an interruptible instruction, the CPU enters the stopped state at the completion of the execution of the current unit of operation. Similarly, in the instruction-step mode, only a single unit of operation is performed, but the unit of operation for the various cases of stopping may be different.

Exceptions to Nullification and Suppression

In certain unusual situations, the result fields of an instruction having a store-type operand are changed in spite of the occurrence of an exception which would normally result in nullification or suppression. These situations are exceptions to the general rule that the operation is treated as a no-operation when an exception requiring nullification or suppression is recognized. Each of these situations may result in the turning on of the change bit associated with the store-type operand, even though the final result in storage may appear unchanged. Depending on the particular situation, additional effects may be observable. The extent of these effects is described along with each of the situations.

All of these situations are limited to the extent that a store access does not occur and the change bit is not set when the store access is prohibited. For the CPU, a store access is prohibited whenever an access exception exists for that access, or whenever an exception exists which is of higher priority than the priority of an access exception for that access.

When, in these situations, an interruption for an exception requiring suppression occurs, the instruction address in the old PSW designates the next sequential instruction. When an interruption for an exception requiring nullification occurs, the instruction address in the old PSW designates the instruction causing the exception even though partial results may have been stored.

Storage Change and Restoration for DAT-Associated Access Exceptions

In this section, the term “DAT-associated access exceptions” is used to refer to those exceptions which may occur as part of the dynamic-address-translation process. These exceptions are ASCE-type, region-first translation, region-second translation, region-third translation,
segment translation, page translation, translation specification, and addressing due to a DAT-table entry being designated at a location that is not available in the configuration. The first six of these exceptions normally cause nullification, and the last two normally cause suppression. Protection exceptions, including those due to page protection, are not considered to be DAT-associated access exceptions.

For DAT-associated access exceptions, on some models, channel programs may observe the effects on storage as described in the following case.

When, for an instruction having a store-type operand, a DAT-associated access exception is recognized for any operand of the instruction, that portion, if any, of the store-type operand which would not cause an exception may be changed to an intermediate value but is then restored to the original value.

The accesses associated with storage change and restoration for DAT-associated access exceptions are only observable by channel programs and are not observable by other CPUs in a multiprocess configuration. Except for instructions which are defined to have multiple-access operands, the intermediate value, if any, is always equal to what would have been the final value if the DAT-associated access exception had not occurred.

Programming Notes:

1. Storage change and restoration for DAT-associated access exceptions occur in two main situations:
   a. The exception is recognized for a portion of a store-type operand which crosses a page boundary, and the other portion has no access exception.
   b. The exception is recognized for one operand of an instruction having two storage operands (for example, an SS-format instruction or MOVE LONG), and the other operand, which is a store-type operand, has no access exception.

2. To avoid letting a channel program observe intermediate operand values due to storage change and restoration for DAT-associated access exceptions (especially when a CCW chain is modified), the CPU program should do one of the following:
   a. Operate on one storage page at a time
   b. Perform preliminary testing to ensure that no exceptions occur for any of the required pages
   c. Operate with DAT off

Modification of DAT-Table Entries

When a valid and attached DAT-table entry is changed to a value which would cause an exception, and when, before the TLB is cleared of entries which qualify for substitution for that entry, an attempt is made to refer to storage by using a virtual address requiring that entry for translation, the contents of any fields due to be changed by the instruction are unpredictable. Results, if any, associated with the virtual address whose DAT-table entry was changed may be placed in those real locations originally associated with the address. Furthermore, it is unpredictable whether or not an interruption occurs for an access exception that was not initially applicable. On some machines, this situation may be reported by means of an instruction-processing-damage machine check with the delayed-access-exception bit also indicated.

Trial Execution for Editing Instructions and Translate Instruction

For the instructions EDIT, EDIT AND MARK, and TRANSLATE, the portions of the operands that are actually used in the operation may be established in a trial execution for operand accessibility that is performed before the execution of the instruction is started. This trial execution consists in an execution of the instruction in which results are not stored. If the first operand of TRANSLATE or either operand of EDIT or EDIT AND MARK is changed by another CPU or by a channel program, after the initial trial execution but before completion of execution, the contents of any fields due to be changed by the instruction are unpredictable. Furthermore, it is unpredictable whether or not an interruption occurs for an access exception that was not initially applicable.
Authorization Mechanisms

Most of the authorization mechanisms that are described in this section permit the control program to establish the degree of function provided to a particular semiprivileged program. These authorization mechanisms are intended for use by programs considered to be semiprivileged, that is, programs that are executed in the problem state but which may be authorized to use additional capabilities. With these authorization controls, a hierarchy of programs may be established, with programs at a higher level having a greater degree of privilege or authority than programs at a lower level. The range of functions available at each level, and the ability to transfer control from a lower to a higher level, are specified in tables which are managed by the control program. When the linkage stack is used, a nonhierarchical transfer of control also can be specified.

A semiprivileged instruction is one which can be executed in the problem state, but which is subject to the control of one or more of the authorization mechanisms described in this section. There are 32 semiprivileged instructions and also the privileged LOAD ADDRESS SPACE PARAMETERS instruction that are controlled by the authorization mechanisms. All of these semiprivileged and privileged instructions are described in Chapter 10, “Control Instructions.”

This section also describes, or refers to descriptions of, authorization mechanisms that are available when the ASN-and-LX-reuse facility is installed. These mechanisms do not apply to particular programs or provide for a hierarchy of programs.

The instructions controlled by the authorization mechanisms are listed in Figure 5-6 on page 5-29. The figure also shows additional authorization mechanisms that do not control specifically semiprivileged instructions; they control implicit access-register translation (access-register translation as part of an instruction making a storage reference) and also access-register translation in the LOAD REAL ADDRESS, STORE REAL ADDRESS, TEST ACCESS, and TEST PROTECTION instructions and a special form of access-register translation in the BRANCH IN SUBSPACE GROUP instruction. These additional mechanisms (the extended authorization index, ALE sequence number, and ASTE sequence number) are described in “Access-Register-Specified Address Spaces” on page 5-44 and the ASTE instance number is further described in “ASN-Second-Table-Entry Sequence Number” on page 3-18. The figure also shows the LSTE sequence number, which may be used in PC-number translation in the PROGRAM CALL instruction and is described in “ASN-and-LX-Reuse Control (R)” on page 5-32.

Mode Requirements

Most of the semiprivileged instructions can be executed only with DAT on. Basic PROGRAM CALL, PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, are valid only in the primary-space mode. (Basic PROGRAM CALL is the PROGRAM CALL operation when the linkage stack is not used. When the linkage stack is used, the PROGRAM CALL operation is called stacking PROGRAM CALL). MOVE TO PRIMARY and MOVE TO SECONDARY are valid only in the primary-space and secondary-space modes. BRANCH AND STACK, stacking PROGRAM CALL, and PROGRAM RETURN are valid only in the primary-space and access-register modes. EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE are valid only in the primary-space, access-register, and home-space modes. When a semiprivileged instruction is executed in an invalid translation mode, a special-operation exception is recognized.

PROGRAM TRANSFER and PROGRAM TRANSFER WITH INSTANCE specify a new value for the problem-state bit in the PSW. If a program in the problem state attempts to execute PROGRAM TRANSFER or PROGRAM TRANSFER WITH INSTANCE and set the supervisor state, a privileged-operation exception is recognized. A privileged-operation exception is also recognized on an attempt to use RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST to set the home-space mode in the problem state.

Extraction-Authority Control

The extraction-authority-control bit is located in bit position 36 of control register 0. In the problem state, bit 36 must be one to allow completion of these instructions:

- EXTRACT PRIMARY ASN
- EXTRACT PRIMARY ASN AND INSTANCE
• EXTRACT SECONDARY ASN
• EXTRACT SECONDARY ASN AND INSTANCE
• INSERT ADDRESS SPACE CONTROL
• INSERT PSW KEY
• INSERT VIRTUAL STORAGE KEY

Otherwise, a privileged-operation exception is recognized. The extraction-authority control is not examined in the supervisor state.

PSW-Key Mask
The PSW-key mask consists of bits 32-47 in control register 3, with the bits corresponding to the values 0-15, respectively, of the PSW key. These bits are used in the problem state to control which keys and entry points are authorized for the program. The PSW-key mask is modified by PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, is modified or loaded by BRANCH AND SET AUTHORITY and PROGRAM CALL, and is loaded by LOAD ADDRESS SPACE PARAMETERS and PROGRAM RETURN. The PSW-key mask is used in the problem state to control the following:

• The PSW-key values that can be set by means of the instruction SET PSW KEY FROM ADDRESS.

• The PSW-key values that are valid for the six move instructions that specify a second access key: MOVE PAGE, MOVE TO PRIMARY, MOVE TO SECONDARY, MOVE WITH KEY, MOVE WITH SOURCE KEY, and MOVE WITH DESTINATION KEY.

• The entry points which can be called by means of PROGRAM CALL. In this case, the PSW-key mask is ANDed with the authorization key mask in the entry-table entry, and, if the result is zero, the program is not authorized.

When an instruction in the problem state attempts to use a key not authorized by the PSW-key mask, a privileged-operation exception is recognized. The same action is taken when an instruction in the problem state attempts to call an entry not authorized by the PSW-key mask. The PSW-key mask is not examined in the supervisor state, all keys and entry points being valid.

Secondary-Space Control
Bit 37 of control register 0 is the secondary-space-control bit. This bit provides a mechanism whereby the control program can indicate whether or not the secondary region-first table, region-second table, region-third table or segment table has been established. Bit 37 may be required to be one to allow completion of SET ADDRESS SPACE CONTROL FAST and must be one to allow completion of these instructions:

• MOVE TO PRIMARY
• MOVE TO SECONDARY
• SET ADDRESS SPACE CONTROL

Otherwise, a special-operation exception is recognized. The secondary-space control is examined in both the problem and supervisor states.

Subsystem-Linkage Control
Bit 192 of the primary ASN-second-table entry is the subsystem-linkage-control bit. The subsystem-linkage control must be one to allow completion of these instructions:

• PROGRAM CALL
• PROGRAM TRANSFER
• PROGRAM TRANSFER WITH INSTANCE

Otherwise, a special-operation exception is recognized. The subsystem-linkage control is examined in both the problem and supervisor states and controls both the space-switching and current-primary versions of the instructions.

ASN-Translation Control
Bit 44 of control register 14 is the ASN-translation-control bit. This bit provides a mechanism whereby the control program can indicate whether ASN translation may occur while a particular program is being executed. Bit 44 must be one to allow completion of these instructions:

• LOAD ADDRESS SPACE PARAMETERS
• SET SECONDARY ASN
• SET SECONDARY ASN WITH INSTANCE
• PROGRAM CALL with space switching
• PROGRAM RETURN with space switching and also when the restored secondary ASN is not equal to the restored primary ASN
• PROGRAM TRANSFER with space switching
• PROGRAM TRANSFER WITH INSTANCE with space switching

Otherwise, a special-operation exception is recognized. The ASN-translation control is examined in both the problem and supervisor states.
ASN-translation control is examined by PROGRAM CALL even though PROGRAM CALL obtains the address of the ASN-second-table entry directly from the entry-table entry instead of by performing ASN translation.

Authorization Index
The authorization index is contained in bit positions 32-47 of control register 4. The authorization index is associated with the primary address space and is loaded along with the PASN when PROGRAM CALL with space switching, PROGRAM RETURN with space switching, PROGRAM TRANSFER with space switching, PROGRAM TRANSFER WITH INSTANCE with space switching, or LOAD ADDRESS SPACE PARAMETERS is executed. The authorization index is used to determine whether a program is authorized to establish a particular address space. A program may be authorized to establish the address space as a secondary-address space, as a primary-address space, or both. The authorization index is examined in both the problem and supervisor states.

Associated with each address space is an authority table. The authorization index is used to select an entry in the authority table. Each entry contains two bits, which indicate whether the program with that authorization index is permitted to establish the address space as a primary address space, as a secondary address space, or both.

The instructions SET SECONDARY ASN with space switching, and SET SECONDARY ASN WITH INSTANCE with space switching, and the instruction PROGRAM RETURN when the restored primary ASN is not equal to the restored primary ASN, use the authorization index to test the secondary-authority bit in the authority-table entry to determine if the address space can be established as a secondary address space. The tested bit must be one; otherwise, a secondary-authority exception is recognized.

The instruction PROGRAM CALL with space switching causes a new authorization index to be loaded from the ASN-second-table entry. This permits the program which is called to be given an authorization index which authorizes it to access more or different address spaces than those authorized for the calling program. The instructions PROGRAM RETURN with space switching, PROGRAM TRANSFER with space switching, and PROGRAM TRANSFER WITH INSTANCE with space switching restore the authorization index that is associated with the returned-to address space.

The secondary-authority bit in the authority-table entry may also be used, along with the extended authorization index, to determine if the program is authorized to use an access-list entry in access-register translation. This is described in “Access-Register-Specified Address Spaces” on page 5-44.

Instructions and Controls Related to ASN-and-LX Reuse
This section describes instructions and controls that are provided when the ASN-and-LX-reuse facility is installed. The controls apply in both the problem and the supervisor states.

Instructions: The ASN-and-LX-reuse facility provides the following instructions:
- EXTRACT PRIMARY ASN AND INSTANCE
- EXTRACT SECONDARY ASN AND INSTANCE
- PROGRAM TRANSFER WITH INSTANCE
- SET SECONDARY ASN WITH INSTANCE
- PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE are similar to PROGRAM TRANSFER and SET SECONDARY ASN, respectively. The space-switching forms of PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE compare an ASTE instance number (ASTEIN) specified by the program in general register 1 to the ASTEIN in the ASN-second-table entry for the address space to which the space switch is to occur. An ASTE-instance exception is recognized if the two ASTEINs are not equal.
**ASN-and-LX-Reuse Control:** Bit 44 of control register 0 is the ASN-and-LX-reuse control. This bit is used to control both ASN reuse and LX reuse, as described below.

**ASN Reuse:** When the ASN-and-LX-reuse control is one, PROGRAM RETURN and LOAD ADDRESS SPACE PARAMETERS, when they perform ASN translation of either a primary ASN or a secondary ASN, compare a specified ASTEIN to the ASTEIN in the ASN-second-table entry located by the ASN translation. The two ASTEINs must be equal; otherwise, PROGRAM RETURN recognizes an ASTE-instance exception, and LOAD ADDRESS SPACE PARAMETERS sets condition code 1 or 2. For PROGRAM RETURN, the specified ASTEIN is in the linkage-stack program-call state entry being unstacked. For LOAD ADDRESS SPACE PARAMETERS, the specified ASTEIN is in the first operand of the instruction.

Bit 31 of word 1 of the ASN-second-table entry is the reusable-ASN bit. When the ASN-and-LX-reuse control is one, the space-switching forms of PROGRAM TRANSFER and SET SECONDARY ASN recognize a special-operation exception if the reusable-ASN bit is one in the ASN-second-table entry located by ASN translation. These instructions do not make a comparison to the ASTEIN in the ASN-second-table entry.

Bit 30 of word 1 of the ASN-second-table entry is the controlled-ASN bit. Regardless of the value of the ASN-and-LX-reuse control, PROGRAM TRANSFER WITH INSTANCE and SET SECONDARY ASN WITH INSTANCE recognize a special-operation exception if the controlled-ASN bit is one in the ASN-second-table entry located by ASN translation and the CPU is in the problem state at the beginning of the operation.

A more detailed description of ASN reuse is given in "ASN-Second-Table-Entry Instance Number and ASN Reuse" on page 3-19.

**LX Reuse:** When the ASN-and-LX-reuse control is one, the linkage table is replaced by a linkage first table and, for each valid linkage-first-table entry, a linkage second table.

The second word of the linkage-second-table entry used contains an LSTE sequence number (LSTESN), and PROGRAM CALL recognizes an LSTE-sequence exception if this LSTESN is nonzero and not equal to an LSTESN specified by the program in general register 15.

A more detailed description of LX reuse is given in "ASN-and-LX-Reuse Control (R)" on page 5-32.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit AR transl. BAKR BSA-ba BSA-ra BSG EPAR</td>
<td>SO-PA SO-PSAH SO-PSAH</td>
<td>0.37 (14.44)</td>
<td>0.36 (3.32-3.47)</td>
<td>Q</td>
<td>EA ALQ ASQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPAIR EREG ESEG ESAR ESAIR ESTA</td>
<td>SO-PAH SO-PAH SO-PAH SO-PAH SO-PAH SO-PAH</td>
<td></td>
<td></td>
<td></td>
<td>Q</td>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IAC IPK IVSK LASP LRA LRAG</td>
<td>SO-PSAH SO-PSAH</td>
<td></td>
<td></td>
<td></td>
<td>SO Q</td>
<td>CCA CCA CCA CCA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSTA MVCDK MVCP MVC SVS MVCS MVCSK bPC-cp</td>
<td>SO-PAH SO-PS SO-PS</td>
<td></td>
<td></td>
<td></td>
<td>Q Q Q Q</td>
<td>ASQ ASQ ASQ ASQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sPC-cp bPC-ss sPC-ss PR-cp PR-ss PT-cp</td>
<td>SO-PAH SO-PAH SO-PAH SO-PAH SO-PAH SO-PAH</td>
<td></td>
<td></td>
<td></td>
<td>SO SO SO SO SO SO</td>
<td>SASS SASS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT-ss11 PTI-cp PTI-ss12 RP SAC SACF</td>
<td>SO-PAH SO-PAH SO-PAH SO-PAH SO-PAH SO-PAH</td>
<td></td>
<td></td>
<td></td>
<td>SO SO SO SO SO SO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAR TPROT</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-6. Summary of Authorization Mechanisms
Explanation for Summary of Authorization Mechanisms:

1. The PSW-key mask is ANDed with the authorization key mask in the entry-table entry.
2. The exception is recognized on an attempt to set the supervisor state when in the problem state.
3. The exception is recognized on an attempt to set the home-space mode when in the problem state.
4. ASN translation is performed for the new SASN, and the exception may be recognized, only when the new SASN is not equal to the new PASN.
5. Secondary authority is checked for the new SASN, and the exception may be recognized, only when the new SASN is not equal to the new PASN.
6. Subsystem-linkage control is bit 192 of the primary ASN-second-table entry.
7. ALE sequence number is bits 8-15 of the access-list entry.
8. ASTE sequence number is bits 160-191 of the ASN-second-table entry.
9. ASTE instance number is bits 352-383 of the ASN-second-table entry.
10. LSTE sequence number is bits 32-63 of the linkage-second-table entry.
11. Special-operation exception is recognized if ASN-and-LX-reuse control, bit 44 of control register 0, is one and reusable-ASN bit, bit 31 of word 1 of the ASN-second-table entry, is one.
12. Special-operation exception is recognized if controlled-ASN bit, bit 30 of word 1 of the ASN-second-table entry, is one and the CPU is in the problem state at the beginning of the operation.
13. Whether the exception is recognized is unpredictable.

A Access-register translation occurs only in the access-register mode.
AIN ASTE-instance exception.
ALQ ALE-sequence exception.

ASQ ASTE-sequence exception.
bPC Basic (nonstacking) PROGRAM CALL.
CC Test results in setting a condition code.
CCA Test results in setting a condition code. The test occurs only in the access-register mode.
CRx.y Control register x, bit position y.
EA Extended-authority exception.
LSQ LSTE-sequence exception.
P Privileged-operation exception for privileged instruction.
PA Primary-authority exception.
PASA Primary-authority exception or secondary-authority exception.
Q Privileged-operation exception for semi-privileged instruction. Authority checked only in the problem state.
SA Secondary-authority exception.
SO Special-operation exception.
SO-P CPU must be in the primary-space mode; special-operation exception if the CPU is in the secondary-space, access-register, home-space, or real mode.
SO-PA CPU must be in the primary-space or access-register mode; special-operation exception if the CPU is in the secondary-space, home-space, or real mode.
SO-PAH CPU must be in the primary-space, access-register, or home-space mode; special-operation exception if the CPU is in the secondary-space or real mode.
SO-PS CPU must be in the primary-space or secondary-space mode; special-operation exception if the CPU is in the home-space, access-register, or real mode.
SO-PSAH CPU must be in the primary-space, secondary-space, access-register, or home-space mode; special-operation exception if the CPU is in the real mode.
sPC Stacking PROGRAM CALL.
When bit 57 of control register 1 is one, a space-switch event is recognized. The operation is completed.

When bit 57 of control register 1 or 13 is one and the instruction space is changed to or from the home address space, a space-switch event is recognized. The operation is completed.

**PC-Number Translation**

PC-number translation is the process of translating a PC number to locate an entry-table entry as part of the execution of the PROGRAM CALL instruction. When the ASN-and-LX-reuse facility is not installed or is not enabled, the PC number is 20 bits in bit positions 44-63 of the effective address used by PROGRAM CALL. When the ASN-and-LX-reuse facility is installed and enabled, the PC number is 20 bits in bit positions 44-63 of the effective address if bit 44 of the address is zero, or it is 32 bits in bit positions 32-63 of the address if bit 44 of the address is one. The facility is enabled if the ASN-and-LX-reuse control, bit 44 of control register 0, is one. The case when the facility is installed and enabled is referred to simply by saying that ASN-and-LX reuse is enabled.

To perform the translation of a PC number to locate an entry-table entry, the PC number is divided into two fields: a linkage index (LX) and an entry index (EX). The entry index is always the rightmost eight bits of the PC number. When ASN-and-LX reuse is not enabled, the leftmost 12 bits of the 20-bit PC number are the linkage index. In this case, the effective address has the following format:

**Effective Address when ASN-and-LX Reuse Is Not Enabled**

```
┌────────/───┐
│ LX │ EX │
└────────/───┘
```

Bit 44 of the effective address has no special meaning and may be zero or one.

When ASN-and-LX reuse is enabled, the linkage index is further divided into a linkage first index (LFX) and a linkage second index (LSX). The linkage second index is always the five bits immediately on the left of the entry index. The size and format of the linkage first index depend on whether the PC number is 20 bits or 32 bits, which in turn depends on whether bit 44 of the effective address is zero or one, respectively. In these cases, the effective address has the following formats:

**Effective Address when ASN-and-LX Reuse Is Enabled and Bit 44 Is Zero**

```
┌─────LFX─────┐
│ 0 │ LFX2 │ LSX │ EX │
└─────/─────┘
```

**Effective Address when ASN-and-LX Reuse Is Enabled and Bit 44 Is One**

```
┌─/─LFX1───┐
│ 1 │ LFX2 │ LSX │ EX │
└─/─/─────┘
```

When ASN-and-LX reuse is enabled and bit 44 of the effective address is zero, the linkage first index is bits 44-50, or bits 45-50 (LFX2) with a zero appended on the left. Thus, the linkage first index is seven bits of which the leftmost bit is always zero. When bit 44 is one, the linkage first index is bits 45-50 (LFX2) with bits 32-43 (LFX1) appended on the left, or 18 bits. However, a linkage first table can contain at most 16,384 entries, and, therefore, the leftmost four bits of the linkage first index, bits 32-35 of the effective address, must always be zeros; otherwise, an LFX-translation exception is recognized.

When ASN-and-LX reuse is not enabled, the translation of a PC number is performed by means of two tables: a linkage table and an entry table. When ASN-and-LX reuse is enabled, the translation is performed by means of three tables: a linkage first table, a linkage second table, and an entry table. All of these tables reside in real storage. The linkage-table designation or linkage-first-table designation resides in another area in storage, called the primary ASN-second-table entry (primary ASTE), whose origin is in control register 5. When there are two levels of tables, The entry table is designated by means of a linkage-table entry. When there are three levels of tables, the linkage second table is designated by means of a linkage-first-table entry, and the entry table is designated by means of a linkage-second-table entry.
Programming Notes:

1. Bit 44 of a PROGRAM CALL effective address specifying a 32-bit PC number is not a numeric part of the PC number.

2. The effective address from which a PC number is derived is subject to the addressing mode in the current PSW. Therefore, bits 0-39 of the effective address in the 24-bit addressing mode, and bits 0-32 in the 31-bit addressing mode, are treated as containing zeros.

PC-Number Translation Control

PC-number translation is controlled by means of the ASN-and-LX-reuse control in control register 0 and a linkage-table designation or linkage-first-table designation in the primary ASN-second-table entry designated by the contents of control register 5.

Control Register 0

ASN-and-LX-Reuse Control (R): Bit 44 of control register 0 is the ASN-and-LX-reuse-control bit and is assigned if the ASN-and-LX-reuse facility is installed. When the bit is one, (1) the PC number is 20 bits if bit 44 of the effective address is zero or 32 bits if bit 44 is one, (2) the PC number is divided into a seven-bit (bit 44 is zero) or 18 bit (bit 44 is one) linkage first index, five-bit linkage second index, and eight-bit entry index, (3) the primary ASN-second-table entry contains a linkage-first-table designation instead of a linkage-table designation, and (4) the PC number is translated by means of a linkage first table, linkage second table, and entry table.

Also when the ASN-and-LX-reuse-control bit is one, if the linkage-second-table entry used during PC-number translation performed by PROGRAM CALL contains a nonzero linkage-second-table-entry sequence number (LSTESN), this LSTESN must be equal to an LSTESN specified in bit positions 0-31 of general register 15; otherwise, the PROGRAM CALL instruction cannot be completed, and an LSTE-sequence exception is recognized.

The ASN-and-LX-reuse control in control register 0 is examined in both the problem and the supervisor states. Other uses of this control, related to the ASN-second-table-entry instance number, are summarized in "ASN-Second-Table-Entry Instance Number and ASN Reuse" on page 3-19.

This use of the LSTESN allows a linkage index associated with a particular LSTESN to be made unusable when the linkage index is reassigned to specify a different (different origin) or conceptually different (different contents, or containing ASNs that designate conceptually different address spaces) entry table. The LX-and-LSTESN combination can be made unusable by changing the LSTESN in the linkage-second-table entry.

Control Register 5

Control register 5 specifies the location of the primary ASN-second-table entry. The register has the following format:

<table>
<thead>
<tr>
<th>R</th>
<th>PASTEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>

Primary-ASTE Origin (PASTEO): Bits 33-57 of control register 5, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the primary ASTE. Bytes 24-27 of the primary ASTE contain the linkage-table designation if the ASN-and-LX-reuse facility is not installed or is not enabled by a one value of the ASN-and-LX-reuse control in control register 0, or they contain the linkage-first-table designation if the facility is installed and enabled.

The linkage-table designation has the following format:

<table>
<thead>
<tr>
<th>V</th>
<th>Linkage-Table Origin</th>
<th>LTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>25 31</td>
</tr>
</tbody>
</table>

Subsystem-Linkage Control (V): Bit 0 of the linkage-table designation is the subsystem-linkage-control bit. Bit 0 must be one to allow completion of these instructions:

- PROGRAM CALL
- PROGRAM TRANSFER
- PROGRAM TRANSFER WITH INSTANCE

Otherwise, a special-operation exception is recognized. The subsystem-linkage control is examined in both the problem and the supervisor states and...
controls both the space-switching and current-
primary versions of the instructions.

**Linkage-Table Origin:** Bits 1-24 of the linkage-
table designation, with seven zeros appended on
the right, form a 31-bit real address that design-
ates the beginning of the linkage table.

**Linkage-Table Length (LTL):** Bits 25-31 of the
linkage-table designation specify the length of the
linkage table in units of 128 bytes, thus making
the length of the linkage table variable in multiples
of 32 four-byte entries. The length of the linkage
table, in units of 128 bytes, is one more than the
value in bit positions 25-31. The linkage-table
length is compared against the leftmost seven bits
of the linkage-index portion of the PC number to
determine whether the linkage index designates
an entry within the linkage table.

The linkage-first-table designation has the fol-
lowing format:

```
+---------+-------------------+---+
| V       | Linkage-First-Table | LFTL |
| 0 1 24 31|
```

**Subsystem-Linkage Control (V):** Bit 0 of the
linkage-first-table designation has the same defi-
nition as bit 0 of the linkage-table designation.

**Linkage-First-Table Origin:** Bits 1-23 of the
linkage-first-table designation, with eight zeros
appended on the right, form a 31-bit real address
that designates the beginning of the linkage first
table.

**Linkage-First-Table Length (LFTL):** Bits 24-31
of the linkage-first-table designation specify the
length of the linkage first table in units of 256
bytes, thus making the length of the linkage first
table variable in multiples of 64 four-byte entries.
The length of the linkage first table, in units of 256
bytes, is one more than the value in bit positions
24-31. When bit 44 of the effective address specify-
ing the PC number is one, the linkage-first-table
length is compared against the leftmost 12 bits of
the linkage-first-index portion of the PC number,
bits 32-43 of the effective address, to determine
whether the linkage first index designates an entry
within the linkage first table. When bit 44 of the
effective address is zero, the linkage-first-table
length is ignored.

**PC-Number Translation Tables**

If the ASN-and-LX-reuse facility is not installed or
is not enabled by a one value of the
ASN-and-LX-reuse control in control register 0, the
PC-number translation process consists in a two-
level lookup using two tables: a linkage table and
an entry table. If the facility is installed and
enabled, the PC-number translation process con-
sists in a three-level lookup using three tables: a
linkage first table, a linkage second table, and an
entry table. All of these tables reside in real
storage.

**Linkage-Table Entries**
The entry fetched from the linkage table has the
following format:

```
+--------+----------+---+
| I      | Entry-Table Origin | ETL |
| 0 1 24 31|
```

The fields in the linkage-table entry are allocated
as follows:

**LX-Invalid Bit (I):** Bit 0 controls whether the
entry table associated with the linkage-table entry
is available.

When the bit is zero, PC-number translation pro-
cceeds by using the linkage-table entry. When the
bit is one, an LX-translation exception is recog-
nized.

**Entry-Table Origin:** Bits 1-25, with six zeros
appended on the right, form a 31-bit real address
that designates the beginning of the entry table.

**Entry-Table Length (ETL):** Bits 26-31 specify
the length of the entry table in units of 128 bytes,
thus making the table variable in multiples of four
32-byte entries. The length of the entry table, in
units of 128 bytes, is one more than the value in bit
positions 26-31. The entry-table length is com-
pared against the leftmost six bits of the entry
index to determine whether the entry index desig-
nates an entry within the entry table.

**Linkage-First-Table Entries**
The entry fetched from the linkage first table has
the following format:
The fields in the linkage-first-table entry are allocated as follows:

**LFX-Invalid Bit (I):** Bit 0 controls whether the linkage second table associated with the linkage-first-table entry is available.

When the bit is zero, PC-number translation proceeds by using the linkage-first-table entry. When the bit is one, an LFX-translation exception is recognized.

**Linkage-Second-Table Origin:** Bits 1-23, with eight zeros appended on the right, form a 31-bit real address that designates the beginning of the linkage second table.

**Programming Note:** The unused field in the linkage-first-table entry, bits 24-31, should be set to zeros. This field is reserved for future extensions, and programs which place nonzero values in this field may not operate compatibly on future machines.

**Linkage-Second-Table Entries**

The entry fetched from the linkage second table has the following format:

<table>
<thead>
<tr>
<th>I</th>
<th>Entry-Table Origin</th>
<th>ETL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>24 31</td>
</tr>
</tbody>
</table>

The fields in the linkage-second-table entry are allocated as follows:

**LSX-Invalid Bit (I):** Bit 0 controls whether the entry table associated with the linkage-second-table entry is available.

When the bit is zero, PC-number translation proceeds by using the linkage-second-table entry. When the bit is one, an LSX-translation exception is recognized.

**Entry-Table Origin:** Bits 1-25, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the entry table.

**Entry-Table Length (ETL):** Bits 26-31 specify the length of the entry table in units of 128 bytes, thus making the table variable in multiples of four 32-byte entries. The length of the entry table, in units of 128 bytes, is one more than the value in bit positions 26-31. The entry-table length is compared against the leftmost six bits of the entry index to determine whether the entry index designates an entry within the entry table.

**Linkage-Second-Table-Entry Sequence Number (LSTESN):** During PC-number translation in PROGRAM CALL when bits 32-63 are not all zeros, bits 32-63 are compared to an LSTESN specified in bit positions 0-31 of general register 15. The comparison must give an equal result; otherwise, an LSTE-sequence exception is recognized. When bits 32-63 are all zeros, bits 0-31 of general register 15 are ignored.

**Programming Note:** The format of bits 0-31 of the linkage-second-table entry is the same as the format of the linkage-table entry.

**Entry-Table Entries**

The format of bits 0-63 of the entry-table entry depends on whether the addressing-mode in effect after the PROGRAM CALL operation is the extended (64-bit) addressing mode or a basic (24-bit or 31-bit) addressing mode. This in turn depends on bits 128 and 129 of the entry-table entry.

Bit 128 of the entry-table entry (T) is the PC-type bit. When bit 128 is zero, PROGRAM CALL is to perform the basic (nonstacking) operation. When bit 128 is one, PROGRAM CALL is to perform the stacking operation.

Bit 129 of the entry-table entry (G) is the entry-extended-addressing-mode bit. In the basic PROGRAM CALL operation, bit 31 of the current PSW, the extended-addressing-mode bit, must equal bit 129; otherwise, a special-operation exception is recognized. In the stacking operation when bit 129 is zero, bit 31 of the current PSW is set to zero, and bit 32 of the PSW, the basic-addressing-mode bit, is set with the value of bit 32 of the entry-table entry (A), the entry-basic-addressing-mode bit. In the stacking operation...
when bit 129 is one, bits 31 and 32 of the current PSW both are set to one. Thus, the basic PROGRAM CALL operation does not switch between the extended and a basic addressing mode but can switch between the 24-bit and 31-bit modes, and the stacking operation can set any addressing mode.

The 32-byte entry-table entry has the following format:

**If Bit 129 is Zero**

0 31

A Entry Instruction Address P

32 63

**If Bit 129 is One**

Entry Instruction Address (Part 1)

0 31

Entry Instruction Address (Part 2) P

32 63

Authorization Key Mask ASN

64 80 95

Entry Key Mask

96 112 127

T G Linkage-Stack Fields

128 130 159

ASTE Origin

160 186 191

Entry Parameter (Part 1)

192 223

Entry Parameter (Part 2)

224 255

The fields in the entry-table entry are allocated as follows:

**Entry Basic Addressing Mode (A):** When bit 129 is zero, bit 32 replaces the basic-addressing-mode bit, bit 32 of the current PSW, as part of the PROGRAM CALL operation. In this case if bit 32 is zero, bits 33-39 must also be zeros; otherwise, a PC-translation-specification exception is recognized. When bit 129 is one, bit 32 is a bit of the entry instruction address, and bit 32 of the PSW remains or is set to one.

**Entry Instruction Address:** When bit 129 is zero, bits 33-62, with 33 zeros appended on the left and a zero appended on the right, form the instruction address which replaces the instruction address in the PSW as part of the PROGRAM CALL operation. When bit 129 is one, bits 0-62, with a zero appended on the right, form the instruction address.

**Entry Problem State (P):** Bit 63 replaces the problem-state bit, bit 15 of the current PSW, as part of the PROGRAM CALL operation.

**Authorization Key Mask:** Bits 64-79 are used to verify whether the program issuing the PROGRAM CALL instruction, when in the problem state, is authorized to call this entry point. The authorization key mask and the current PSW-key mask in control register 3 are ANDed, and the result is checked for all zeros. If the result is all zeros, a privileged-operation exception is recognized. The test is not performed in the supervisor state.

**ASN:** Bits 80-95 specify whether a space-switching (PC-ss) operation or a to-current-primary (PC-cp) operation is to occur. When bits 80-95 are zeros, PC-cp is specified. When bits 80-95 are not all zeros, PC-ss is specified, and the bits are the ASN that replaces the primary ASN.

**Entry Key Mask:** Bits 96-111 may be ORed into or may replace the PSW-key mask in control register 3 as part of the PROGRAM CALL operation, as determined by a bit in bit positions 130-159.

**PC-Type Bit (T):** Bit 128 specifies the basic PROGRAM CALL operation when the bit is zero or the stacking PROGRAM CALL operation when the bit is one.

**Entry Extended Addressing Mode (G):** In the basic PROGRAM CALL operation, bit 129 must match the extended-addressing-mode bit, bit 31 of
the current PSW; otherwise, a special-operation exception is recognized. In the stacking operation, bit 129 replaces bit 31 of the PSW.

**ASTE Origin:** When bits 80-95 are not all zeros, bits 161-185, with six zeros appended on the right, form the 31-bit real ASN-second-table-entry address that should result from applying the ASN-translation process to bits 80-95.

**Entry Parameter:** When bit 129 is zero, bits 224-255 are placed in bit positions 32-63 of general register 4, and bits 0-31 of the register remain unchanged, as part of the PC operation. When bit 129 is one, bits 192-255 are placed in general register 4 as part of the PC operation.

Bits 130-159 are used in connection with the linkage stack and are described in "Linkage-Stack Entry-Table Entries" on page 5-74.

Bits 112-127, 160, and 186-191 are reserved for possible future extensions and should be zeros.

**Programming Note:** The entry parameter is intended to provide the called program with an address which can be depended upon and used as the basis of addressability in locating necessary information which may be environment dependent. The parameter may be appropriately changed for each environment by setting up different entry tables. The alternative — obtaining this information from the calling program — may require extensive validity checking or may present an integrity exposure.

**Table Summary**

Figure 5-7 on page 5-37 presents a summary of information about the tables used in PC-number translation, except for the primary ASN second table.

**PC-Number-Translation Process**

When the ASN-and-LX-reuse facility is not installed or is not enabled, the translation of the PC number is performed by means of a linkage table and entry table. When the ASN-and-LX-reuse facility is installed and enabled, referred to by saying that ASN-and-LX reuse is enabled, the translation of the PC number is performed by means of a linkage first table, linkage second table and entry table. All of these tables reside in real storage. The translation also requires the use of the primary ASN-second-table entry, which also resides in real storage.

For the purposes of PC-number translation, the PC number is divided into two parts: the leftmost part is called the linkage index (LX), and the rightmost eight bits are called the entry index (EX). When ASN-and-LX reuse is not enabled, the PC number is 20 bits, and the linkage index is 12 bits. When ASN-and-LX reuse is enabled, the linkage index also is divided into two parts: the leftmost part is called the linkage first index (LFX), and the rightmost five bits are called the linkage second index (LSX). When ASN-and LX reuse is enabled and bit 44 of the effective address specifying the PC number is zero, the PC number is 20 bits, and the linkage first index is seven bits, or, when bit 44 is one, the PC number is 32 bits, and the linkage first index is 18 bits.

When ASN-and-LX reuse is not enabled, the LX is used to select an entry from the linkage table, the starting address and length of which are specified by the linkage-table designation in the primary ASTE. This entry designates the entry table to be used. The EX field of the PC number is used to select an entry from the entry table.

When ASN-and-LX reuse is enabled, the LFX is used to select an entry from the linkage first table, the starting address and length of which are specified by the linkage-first-table designation in the primary ASTE. This entry designates the linkage second table to be used. The LSX is then used to select an entry from the linkage second table, which entry designates the entry table to be used. The EX field of the PC number is then used to select an entry from the entry table.

When ASN-and-LX reuse is enabled, a linkage-second-table-entry sequence number (LSTESN) in general register 15 is compared to the LSTESN in the linkage-second-table entry if the LSTESN in the entry is nonzero.

When, for the purposes of PC-number translation, accesses are made to main storage to fetch entries from the primary ASTE, linkage table, linkage first table, linkage second table, and entry table, key-controlled protection does not apply.

The PC-number-translation process is shown in Figure 5-8 on page 5-38 for when
<table>
<thead>
<tr>
<th>Table</th>
<th>Boundary Size (Bytes)</th>
<th>Entry Size (Bytes)</th>
<th>Unit Size</th>
<th>Maximum Number of Units</th>
<th>Maximum Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bytes</td>
<td>Entries</td>
<td></td>
</tr>
<tr>
<td>Linkage</td>
<td>128</td>
<td>4</td>
<td>128</td>
<td>32</td>
<td>16384, 4096</td>
</tr>
<tr>
<td>Linkage First</td>
<td>256</td>
<td>4</td>
<td>256</td>
<td>64</td>
<td>65536, 16384</td>
</tr>
<tr>
<td>Linkage Second</td>
<td>256</td>
<td>8</td>
<td>256</td>
<td>32</td>
<td>256, 32</td>
</tr>
<tr>
<td>Entry</td>
<td>64</td>
<td>32</td>
<td>128</td>
<td>4</td>
<td>8192, 256</td>
</tr>
</tbody>
</table>

**Explanation:**

1. A 20-bit PC number can specify an entry only within the first unit. Bits 0-3 of a 32-bit PC number specify entries beyond the end of the largest possible table.

*Figure 5-7. Summary of PC-Number-Translation Tables*

ASN-and-LX-reuse is not enabled and in Figure 5-9 on page 5-39 for when it is enabled.
Linkage-Table Designation
in Primary ASTE

V LTO LTL
(x128)

20-Bit
PC Number
LX EX
(x4) (x32)

Linkage Table

Entry Table

** A EIA P AKM ASN EKM
TG L.-S. Fields ASTE Origin EP

R: Address is real
**: First word and A of ETE are bits 0-32 of EIA if G is one.

Figure 5-8. PC-Number Translation when ASN-and-LX Reuse Is Not Enabled
Figure 5-9. PC-Number Translation when ASN-and-LX Reuse Is Enabled
Obtaining the Linkage-Table or Linkage-First-Table Designation

The linkage-table or linkage-first-table designation is obtained from bytes 24-27 of the primary ASN-second-table entry, the starting address of which is specified by the contents of control register 5.

The 31-bit real address of the linkage-table or linkage-first-table designation is obtained by appending six zeros on the right to the primary-ASTE origin, bits 33-57 of control register 5, and adding 24. The addition cannot cause a carry into bit position 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All four bytes of the linkage-table or linkage-first-table designation appear to be fetched concurrently from the primary ASTE as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the linkage-table or linkage-first-table designation designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed. Besides the linkage-table or linkage-first-table designation, no other field in the primary ASTE is examined.

Linkage-Table Lookup

When the ASN-and-LX-reuse facility is not installed or the ASN-and-LX-reuse-control bit, bit 44 of control register 0, is zero, the field fetched from the primary ASTE is a linkage-table designation. The linkage-index (LX) portion of the PC number, in conjunction with the linkage-table origin, is used to select an entry from the linkage table.

The 31-bit real address of the linkage-table entry is obtained by appending seven zeros on the right to the contents of bit positions 1-24 of the linkage-table designation and adding the linkage index, with two rightmost and 17 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{31} - 1$ to 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the linkage-table-lookup process, the leftmost seven bits of the linkage index are compared against the linkage-table length, bits 25-31 of the linkage-table designation, to establish whether the addressed entry is within the linkage table. If the value in the linkage-table-length field is less than the value of the seven leftmost bits of the linkage index, an LX-translation exception is recognized.

Bit 0 of the linkage-table entry specifies whether the entry table corresponding to the linkage index is available. This bit is inspected, and, if it is one, an LX-translation exception is recognized.

When no exceptions are recognized in the process of linkage-table lookup, the entry fetched from the linkage table designates the origin and length of the corresponding entry table.

Linkage-First-Table Lookup

When the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse-control bit, bit 44 of control register 0, is one, the field fetched from the primary ASTE is a linkage-first-table designation. The linkage-first-index (LFX) portion of the PC number, in conjunction with the linkage-first-table origin, is used to select an entry from the linkage first table.

When bit 44 of the effective address of PROGRAM CALL is zero, the PC number is 20 bits, the linkage first index is bits 44-50 of the effective address, and the 31-bit real address of the linkage-first-table entry is obtained by appending eight zeros on the right to the contents of bit positions 1-23 of the linkage-first-table designation and adding the linkage first index, with two rightmost and 22 leftmost zeros appended.

When bit 44 of the effective address of PROGRAM CALL is one, the PC number is 32 bits, the linkage first index is bits 32-43 of the effective address appended on the left of bits 45-50 of the effective address, and the 31-bit real
address of the linkage-first-table entry is obtained by appending eight zeros on the right to the contents of bit positions 1-23 of the linkage-first-table designation and adding the linkage first index, with two rightmost and 11 leftmost zeros appended.

When bit 44 of the effective address is one and a carry into bit position 0 occurs during the addition to form the address of a linkage-first-table entry, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{31} - 1$ to 0.

When bit 44 of the effective address is either zero or one, the 31-bit address of the linkage-first-table entry is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the linkage-first-table-lookup process when bit 44 of the effective address is one, the leftmost 12 bits of the linkage first index are compared against the linkage-first-table length, bits 28-31 of the linkage-first-table designation, to establish whether the addressed entry is within the linkage first table. For this comparison, the linkage-first-table length is extended with eight zero bits on the left. If the value of the extended linkage-first-table length is less than the value of the 12 leftmost bits of the linkage first index, an LFX-translation exception is recognized.

All four bytes of the linkage-first-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the linkage-second-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the linkage-first-table entry specifies whether the linkage second table corresponding to the linkage first index is available. This bit is inspected, and, if it is one, an LFX-translation exception is recognized.

When no exceptions are recognized in the process of linkage-first-table lookup, the entry fetched from the linkage first table designates the origin of the corresponding linkage second table.

**Linkage-Second-Table Lookup**

When the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse-control bit, bit 44 of control register 0, is one, a linkage-second-table lookup is performed after a linkage-first-table entry has been fetched. The linkage-second-index (LSX) portion of the PC number, in conjunction with the linkage-second-table origin, is used to select an entry from the linkage second table.

The 31-bit real address of the linkage-second-table entry is obtained by appending eight zeros on the right to the linkage-second-table origin, bits 1-23 of the linkage-first-table entry, and adding the linkage second index, with three rightmost and 23 leftmost zeros appended. A carry into bit position 0 cannot occur during this addition. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

All eight bytes of the linkage-second-table entry appear to be fetched concurrently as observed by other CPUs. The fetch access is not subject to protection. When the storage address which is generated for fetching the linkage-second-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the linkage-second-table entry specifies whether the entry table corresponding to the linkage second index is available. This bit is inspected, and, if it is one, an LSX-translation exception is recognized.

When no exceptions are recognized in the process of linkage-second-table lookup, the entry fetched from the linkage second table designates the origin of the corresponding entry table. The linkage-second-table entry contains a linkage-second-table-entry sequence number that may be used to test the correctness of the use of the linkage index.

**Linkage-Second-Table-Entry-Sequence-Number Comparison**

The linkage-second-table entry contains a linkage-second-table-entry sequence number (LSTESN) in bit positions 32-63. If this LSTESN is nonzero, it is compared to an LSTESN in bit positions 0-31 of general register 15, and an LSTE-sequence exception is recognized if the two LSTESNs are not equal.
Entry-Table Lookup

When the ASN-and-LX-reuse facility is not installed or is not enabled, the entry-table entry is located from a linkage-table entry. When the ASN-and-LX-reuse facility is installed and enabled, the entry-table entry is located from a linkage-second-table entry.

The entry-index (EX) portion of the PC number, in conjunction with the entry-table origin contained in the linkage-table or linkage-second-table entry, is used to select an entry from the entry table.

The 31-bit real address of the entry-table entry is obtained by appending six zeros on the right to the entry-table origin bits 1-25 of the linkage-table or linkage-second-table entry, and adding the entry index, with five rightmost and 18 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the table to wrap from $2^{31} - 1$ to 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the entry-table-lookup process, the six leftmost bits of the entry index are compared against the entry-table length, bits 26-31 of the linkage-table or linkage-second-table entry, to establish whether the addressed entry is within the table. If the value in the entry-table length field is less than the value of the six leftmost bits of the entry index, an EX-translation exception is recognized.

The 32-byte entry-table entry is fetched by using the real address. The fetch of the entry appears to be word concurrent, as observed by other CPUs, with the leftmost word fetched first. The order in which the remaining seven words are fetched is unpredictable. The fetch access is not subject to protection. When the storage address which is generated for fetching the entry-table entry designates a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

The use that is made of the information fetched from the entry-table entry is described in the definition of the PROGRAM CALL instruction.

Recognition of Exceptions during PC-Number Translation

The exceptions which can be encountered during the PC-number-translation process and their priority are described in the definition of the PROGRAM CALL instruction.

Programming Note: The linkage-table or linkage-first-table designation is fetched successfully from the primary ASN-second-table entry regardless of the value of bit 0, the ASX-invalid bit, in the primary ASTE. A one value of this bit may cause an exception to be recognized in other circumstances.

Home Address Space

Facilities are provided which a privileged program, such as the control program, can use to obtain control in and access the home address space of a dispatchable unit (for example, a task).

Each dispatchable unit normally has an address space associated with it in which the control program keeps the principal control blocks that represent the dispatchable unit. This address space is called the home address space of the dispatchable unit. Different dispatchable units may have the same or different home address spaces. When the control program initiates a dispatchable unit, it may set the primary and secondary address spaces equal to the home address space of the dispatchable unit. Thereafter, because of the dispatchable unit's possible use of the PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, or SET SECONDARY ASN instruction, the control program normally cannot depend on either the primary address space or the secondary address space being the home address space when the home address space must be accessed, for example, during the processing by the control program of an interruption. Therefore, the control program normally must take some special action to ensure that the home address space is addressed when it must be accessed. The home-address-space facilities provide an efficient means to take this action.

The home-address-space facilities include:

- The home address-space-control element (HASCE) in control register 13. The HASCE is used by DAT in the same way as the
primary address-space-control element (PASCE) in control register 1 and the secondary address-space-control element (SASCE) in control register 7.

- Home-space mode, which results when DAT is on and the address-space control, PSW bits 16 and 17, has the value 11 binary. When the CPU is in the home-space mode, instruction and logical addresses are home virtual addresses and are translated by DAT by means of the HASCE.

- The ability of the RESUME PROGRAM, SET ADDRESS SPACE CONTROL, and SET ADDRESS SPACE CONTROL FAST instructions to set the home-space mode in the supervisor state, and the ability of the INSERT ADDRESS SPACE CONTROL instruction to return an indication of the home-space mode.

- The home space-switch-event control, bit 57 of control register 13.

- Recognition of a space-switch event upon completion of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction if the CPU was in the home-space mode before or after the operation but not both before and after the operation, if any of the following is true: (1) the primary space-switch-event control, bit 57 of control register 1, is one, (2) the home space-switch-event control is one, or (3) a PER event is to be indicated.

The space-switch event can be used to enable or disable PER or tracing when fetching of instructions begins or ends in particular address spaces.

---

**Access-Register Introduction**

Many of the functions related to access registers are described in this section and in "Subroutine Linkage without the Linkage Stack" on page 5-11, "Access-Register Translation" on page 5-51, and "Sequence of Storage References" on page 5-88. Additionally, translation modes and access-list-controlled protection are described in Chapter 3, "Storage" and during the store-status operation are described in Chapter 4, "Control"; interruptions are described in Chapter 6, "Interruptions"; instructions are described in Chapter 7, "General Instructions," and Chapter 10, "Control Instructions"; the handling of access registers during a machine-check interruption and the programmed validation of the access registers are described in Chapter 11, "Machine-Check Handling"; and the alter-and-display controls for access registers are described in Chapter 12, "Operator Facilities."

---

**Summary**

These major functions are provided:

- A maximum of 16 address spaces, including the instruction space, for immediate and simultaneous use by a semiprivileged program; the address spaces are specified by 16 registers called access registers.

- Instructions for examining and changing the contents of the access registers.

In addition, control and authority mechanisms are incorporated to control these functions.

Access registers allow a sequence of instructions, or even a single instruction such as MOVE (MVC) or MOVE LONG (MVCL), to operate on storage operands in multiple address spaces, without the requirement of changing either the translation mode or other control information. Thus, a program residing in one address space can use the complete instruction set to operate on data in that address space and in up to 15 other address spaces, and it can move data between any and all pairs of these address spaces. Furthermore, the program can change the contents of the access registers in order to access still other address spaces.

The instructions for examining and changing access-register contents are unprivileged and are described in Chapter 7, "General Instructions." They are:

- COPY ACCESS
- EXTRACT ACCESS
- LOAD ACCESS MULTIPLE
- LOAD ADDRESS EXTENDED
- SET ACCESS
- STORE ACCESS MULTIPLE
The privileged PURGE ALB instruction and COMPARE AND SWAP AND PURGE instruction are used in connection with access registers and are described in Chapter 10, “Control
Instructions.”

Access registers specify address spaces when the CPU is in the access-register mode. The SET ADDRESS SPACE CONTROL and SET ADDRESS SPACE CONTROL FAST instructions allow setting of the access-register mode, and the INSERT ADDRESS SPACE CONTROL instruction provides an indication of the access-register mode. The stacking PROGRAM CALL, PROGRAM RETURN, and RESUME PROGRAM instructions also allow setting of the access-register mode. All of these instructions are described in Chapter 10, “Control Instructions.”

Access registers are used in a special way by the BRANCH IN SUBSPACE GROUP instruction. The use of access registers by that instruction is described in detail only in the definition of the instruction in Chapter 10, “Control Instructions.” However, “Subspace-Group Tables” on page 5-64 describes the use of the dispatchable-unit control table and the extended ASN-second-table entry by BRANCH IN SUBSPACE GROUP.

Access-Register Functions

Access-Register-Specified Address Spaces
The CPU includes sixteen 32-bit access registers numbered 0-15. In the access-register mode, which results when DAT is on and PSW bits 16 and 17 are 01 binary, an instruction B or R field that is used to specify the logical address of a storage operand designates not only a general register but also an access register. The designated general register is used in the ordinary way to form the logical address of the storage operand. The designated access register is used to specify the address space to which the logical address is relative. The access register specifies the address space by specifying an address-space-control element for the address space, and this address-space-control element is used by DAT to translate the logical address. An access register specifies an address-space-control element in an indirect way, not by containing the address-space-control element.

An access register may specify the primary or secondary address-space-control element in control register 1 or 7, respectively, or it may specify an address-space-control element contained in an ASN-second-table element entry. In the latter case, the access register designates an entry in a table called an access list, and the designated access-list entry in turn designates the ASN-second-table entry.

The process of using the contents of an access register to obtain an address-space-control element for use by DAT is called access-register translation (ART). This is depicted in Figure 5-10.

Figure 5-10. Use of Access Registers

An access register is said to specify an AR-specified address space by means of an AR-specified address-space-control element. The virtual addresses in an AR-specified address space are called AR-specified virtual addresses.

In the access-register mode, whereas all storage-operand addresses are AR-specified virtual, instruction addresses are primary virtual.

Designating Access Registers: In the access-register mode, an instruction B or R field designates an access register, for use in access-register translation, under the following conditions:

- The field is a B field which designates a general register containing a base address. The base address is used, along with a displacement (D) and possibly an index (X), to form the logical address of a storage operand.
• The field is an R field which designates a
general register containing the logical address
of a storage operand.

For example, consider the following instruction:

\[ \text{MVC } 0(L,1),0(2) \]

The second operand, of length L, is to be moved
to the first-operand location. The logical address
of the second operand is in general register 2, and
that of the first-operand location in general register
1. The address space containing the second
operand is specified by access register 2, and that
containing the first-operand location by access
register 1. These two address spaces may be dif-
ferent address spaces, and each may be different
from the current instruction address space (the
primary address space).

When PSW bits 16 and 17 are 01, the B2 field of
the LOAD REAL ADDRESS and STORE REAL
ADDRESS instructions designates an access reg-
ister, for use in access-register translation, regard-
less of whether DAT is on or off.

The COMPARE AND FORM CODEWORD and
UPDATE TREE instructions specify storage oper-
ands by means of implicitly designated general
registers and access registers.

The MOVE TO PRIMARY and MOVE TO SEC-
ONDARY instructions specify storage operands by
means of primary virtual and secondary virtual
addresses, and access registers do not apply to
these instructions. An exception is recognized
when either of these instructions is executed in
the access-register mode. The MOVE WITH KEY
instruction can be used in place of MOVE TO
PRIMARY and MOVE TO SECONDARY in the
access-register mode. The MOVE WITH
SOURCE KEY and MOVE WITH DESTINATION
KEY instructions also can be used.

An instruction R field may designate an access
register for other than the purpose of access-
register translation.

The fields which may designate access registers,
whether or not for access-register translation, are
indicated in the summary figure at the beginning
of each instruction chapter.

Obtaining the Address-Space-Control
Element: This section and the following ones
introduce the access-register-translation process and present the concepts related to access lists.

The address-space-control element specified by
an access register is obtained by access-register translation as follows:

• If the access register contains 00000000 hex,
the specified address-space-control element is
the primary address-space-control element
(PASCE), obtained from control register 1.

• If the access register contains 00000001 hex,
the specified address-space-control element is
the secondary address-space-control element
(SASCE), obtained from control register 7.

• If the access register contains any other
value, the specified address-space-control
element is obtained from an
ASN-second-table entry. The contents of the
access register designate an access-list entry that
contains the real origin of the
ASN-second-table entry.

Access register 0 is treated in a special way by
access-register translation; it is treated as con-
taining 00000000 hex, and its actual contents are
not examined. Thus, a logical address specified
by means of a zero B or R field in the access-
register mode is always relative to the primary
address space, regardless of the contents of
access register 0. However, there is one excep-
tion to how access register 0 is treated: the TEST
ACCESS instruction uses the actual contents of
access register 0, instead of treating access reg-
ister 0 as containing 00000000 hex.

The treatment of an access register containing the
value 00000000 hex as designating the current
primary address space allows that address space
to be addressed, in the access-register mode,
without requiring the use of an access-list entry.
This is useful when the primary address space is
changed by a space-switching PROGRAM CALL
(PC-ss), PROGRAM RETURN (PR-ss), or
PROGRAM TRANSFER (PT-ss) instruction. Simi-
larly, the treatment of an access register con-
taining the value 00000001 hex as designating the
secondary address space allows that space to be
addressed after a space-switching operation,
again without requiring the use of an access-list
entry.
The contents of the access registers are not changed by the PROGRAM CALL and PROGRAM TRANSFER instructions. Therefore, an access register containing 00000000 or 00000001 hex may specify a different address space after the execution of PROGRAM CALL or PROGRAM TRANSFER than before the execution. For example, if a space-switching PROGRAM CALL instruction is executed, an access register containing 00000000 hex specifies the old primary address space before the execution and the new primary address space after the execution.

When access-register translation obtains an address-space-control element from an ASN-second-table entry, bit 0 of the entry, the ASX-invalid bit, must be zero; otherwise, an exception is recognized.

**Access Lists:** The access-list entry that is designated by the contents of an access register can be located in either one of two access lists, the dispatchable-unit access list or the primary-space access list. A bit in the access register specifies which of the two access lists contains the designated entry. Both of the access lists reside in real or absolute storage. The locations of the access lists are specified by means of control registers 2 and 5.

Control register 2 contains the origin of a real-storage area called the dispatchable-unit control table. The dispatchable-unit control table contains the designation — the real or absolute origin, and length — of the dispatchable-unit access list.

Control register 5 contains the origin of a real-storage area called the primary ASN-second-table entry. The primary ASN-second-table entry contains the designation of the primary-space access list.

An access list, either the dispatchable-unit access list or the primary-space access list, contains some multiple of eight 16-byte entries, up to a maximum of 1,024 entries.

**Programs and Dispatchable Units:** When discussing access lists, it is necessary to distinguish between the terms "program" and "dispatchable unit." A program is a sequence of instructions and may be referred to as a program module. A program may be a sequence of calling and called programs. A dispatchable unit, which is sometimes called a process or a task, is a unit of work that is performed through the execution of a program by one CPU at a time.

The dispatchable-unit access list is intended to be associated with a dispatchable unit; that is, it is intended that a dispatchable unit have the same dispatchable-unit access list regardless of which program is currently being executed to perform the dispatchable unit. There is no mechanism, except for the LOAD CONTROL instruction, that changes the dispatchable-unit-control-table origin in control register 2.

The primary-space access list is associated with the primary address space that is specified by the primary ASN in control register 4 and the primary address-space-control element in control register 1. The primary-space access list that is available for use by a dispatchable unit changes as the primary address space of the dispatchable unit changes, that is, whenever a program in a different primary address space begins to be executed to perform the dispatchable unit. Whenever a LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, or PROGRAM TRANSFER instruction replaces the primary ASN in control register 4 and the primary address-space-control element in control register 1, it also replaces the primary-ASN-second-table-entry origin in control register 5.

Thus, for a dispatchable unit, the dispatchable-unit access list is intended to be constant (although its entries may be changed, as will be described), and the primary-space access list is a function of which program is being executed, through being a function of the primary address space of the program. Also, all dispatchable units and programs in the same primary address space have the same primary-space access list.

**Access-List-Entry Token:** The contents of an access register are called an access-list-entry token (ALET) since, in the general case, they designate an entry in an access list. An ALET has the following format:

```
| /zerodot/zerodot/zerodot/zerodot/zerodot/zerodot/zerodot/zerodot | P | ALESN | ALEN |
```

The ALET contains a primary-list bit (P) that specifies which access list contains the designated
access-list entry: the dispatchable-unit access list if the bit is zero, or the primary-space access list if the bit is one. The specified access list is called the effective access list.

The ALET also contains an access-list-entry number (ALEN) which, when multiplied by 16, is the number of bytes from the beginning of the effective access list to the designated access-list entry. During access-register translation, an exception is recognized if the ALEN designates an entry that is outside the effective access list or if the leftmost seven bits in the ALET are not all zeros.

The access-list-entry sequence number (ALESN) in the ALET is described in the next section.

The above format of the ALET does not apply when the ALET is 00000000 or 00000001 hex.

An ALET can exist in an access register, in a general register, or in storage, and it has no special protection from manipulation by the problem program. Any program can transfer ALETs back and forth among access registers, general registers, and storage. A called program can save the contents of the access registers in any storage area available to it, load and use the access registers for its own purposes, and then restore the original contents of the access registers before returning to its caller.

**Allocating and Invalidating Access-List Entries:** It is intended that access lists be provided by the control program and that they be protected from direct manipulation by any problem program. This protection may be obtained by means of key-controlled protection or by placing the access lists in real storage not accessible by any problem program by means of DAT.

As determined by a bit in the entry, an access-list entry is either valid or invalid. A valid access-list entry specifies an address space and can be used by a suitably authorized program to access that space. An invalid access-list entry is available for allocation as a valid entry. It is intended that the control program provide services that allocate valid access-list entries and that invalidate previously allocated entries.

Allocation of an access-list entry may consist in the following steps. A problem program passes some kind of identification of an address space to the control program, and it passes a specification of either the dispatchable-unit access list or the primary-space access list. The control program checks, by some means, the authority of the problem program to access the address space. If the problem program is authorized, the control program selects an invalid entry in the specified access list, changes it to a valid entry specifying the subject address space, and returns to the problem program an access-list-entry token (ALET) that designates the allocated entry. The problem program can subsequently place the ALET in an access register in order to access the address space. Later, through the use of the invalidation service of the control program, the access-list entry that was allocated may be made invalid. An exception is recognized during access-register translation if an ALET is used that designates an invalid access-list entry.

It may be that a particular access-list entry is allocated, then invalidated, and then allocated again, this time specifying a different address space than the first time. To guard against erroneous use of an ALET that designates a conceptually wrong address space, an access-list-entry sequence number (ALESN) is provided in both the ALET and the access-list entry. When the control program allocates an access-list entry, it should place the same ALESN in the entry and in the designating ALET that it returns to the problem program. When the control program reallocates an access-list entry, it should change the value of the ALESN. An exception is recognized during access-register translation if the ALESN in the ALET used is not equal to the ALESN in the designated access-list entry.

The ALESN check is a reliability mechanism, not an authority mechanism, because the ALET is not protected from the problem program, and the problem program can change the ALESN in the ALET to any value. Also, this is not a fail-proof reliability mechanism because the ALESN is one byte and its value wraps around after 256 reallocations, assuming that the value is incremented by one for each reallocation.

**Authorizing the Use of Access-List Entries:** Although an access list is intended to be associated with either a dispatchable unit or a primary address space, the valid entries in the list are intended to be associated with the different pro-
grams that are executed, in some order, to perform the work of the dispatchable unit. It is intended that each program be able to have a particular authority that permits the use of only those access-list entries that are associated with the program. The authority being referred to here is represented by a 16-bit extended authorization index (EAX) in control register 8.

Other elements used in the related authorization mechanism are: (1) a private bit in the access-list entry, (2) an access-list-entry authorization index (ALEAX) in the access-list entry, and (3) the authority table.

A program is authorized to use an access-list entry, in access-register translation, if any of the following conditions is met:

1. The private bit in the access-list entry is zero. This condition provides a high-performance means to authorize any and all programs that are executed to perform the dispatchable unit.

2. The ALEAX in the access-list entry is equal to the EAX in control register 8. This condition provides a high-performance means to authorize only particular programs.

3. The EAX selects a secondary bit that is one in the authority table associated with the address space that is specified by the access-list entry. The authority table is locatable in that the access-list entry contains the real origin of the ASN-second-table entry (ASTE) for the address space, and the ASTE contains the real origin of the authority table. This condition provides another means, less well-performing than condition 2, for authorizing only particular programs. However, providing for condition 3 to be met instead of condition 2 can be advantageous because it permits several programs, each executed with a different EAX, all to use a single access-list entry to access a particular address space.

Access-register translation tests for the three conditions in the order indicated by their numbers, and a higher-numbered condition is not tested for if a lower-numbered condition is met. An exception is recognized if none of the conditions is met.

Figure 5-11 shows an example of how the authorization mechanism can be used. In the figure, “PBZ” means that the private bit is zero, and “PBO” means that the private bit is one.

Access List

<table>
<thead>
<tr>
<th>Access List</th>
<th>ASTE for Space 36</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>PBZ</td>
</tr>
<tr>
<td>7</td>
<td>PBO, ALEAX = 5</td>
</tr>
<tr>
<td>9</td>
<td>PBO, ALEAX = 10</td>
</tr>
<tr>
<td>12</td>
<td>PBO, ALEAX = 5</td>
</tr>
</tbody>
</table>

Authority Table

| ASTE for Space 17 | S bit selected by EAX 10 is one. |

Program A

| EAX = 0 |

Program B

| EAX = 5 |

Program C

| EAX = 10 |

Figure 5-11. Example of Authorizing the Use of Access-List Entries

The figure shows an access list — assume it is a dispatchable-unit access list — in which the entries of interest are entries 4, 7, 9, and 12. Each access-list entry contains a private bit, an
ALEAX, and the real origin of the ASTE for an address space. The private bit in entry 4 is zero, and, therefore, the value of the ALEAX in entry 4 is immaterial and is not shown. The private bits in entries 7, 9, and 12 are ones, and the ALEAX values in these entries are as shown. The numbers used to identify the address spaces (36, 25, 62, and 17) are arbitrary. They may be the ASNs of the address spaces; however, ASNs are in no way used in access-register translation. Only the authority table for address space 17 is shown. In it, the secondary bit selected by EAX 10 is one. Assume that no secondary bits are ones in the authority tables for the other spaces.

The figure also shows a sequence of three programs, named A, B, and C, that is executed to perform the work of the dispatchable unit associated with the access list. These programs may be in the same or different address spaces. The EAX in control register 8 when each of these programs is executed is 0, 5, and 10, respectively.

Each of programs A, B, and C can use access-list entry (ALE) 4 to access address space 36 since the private bit in ALE 4 is zero. Program B can use ALE 7 to access space 25 because the ALEAX in the ALE equals the EAX for the program, and no other program can use this ALE. Similarly, only program C can use ALE 9. Program B can use ALE 12 because the ALEAX and EAX are equal, and program C can use it because C’s EAX selects a secondary bit that is one in the authority table for space 17.

The example would be the same if programs A, B, and C were all in the same address space and the access list were the primary-space access list for that space.

An ALE in which the private bit is zero may be called public because the ALE can be used by any program, regardless of the value of the current EAX. An ALE in which the private bit is one may be called private because the ability of a program to use the ALE depends on the current EAX.

Notes on the Authorization Mechanism: An access list is a kind of capability list, in the sense in which the word “capability” is used in computer science. It is up to the control program to formulate the policies that are used to allocate entries in an access list, and the programmed authorization checking required during allocation may be very complex and lengthy. After a valid entry has been made in an access list, the access-register-translation process enforces the control-program policies in a well-performing way by means of the authorization mechanism described above.

Using access lists has an advantage over using only ASNs and authority tables. For example, assume that an access register could contain an ASN and that access-register translation would do ASN translation of the ASN and then use the EAX to test the authority table. This would make the EAX relevant to all existing address spaces, and, therefore, it would make the management of EAXs and their assignment to programs more difficult. With the actual definitions of the ALET and access-register translation, an EAX is relevant to only the address spaces that are represented in the current dispatchable-unit and primary-space access lists. Also, since ASN translation is not done as a part of access-register translation, the number of concurrently existing address spaces, as represented by ASN-second-table entries, can be greater than the number of available ASNs (64K).

The entry-table entry and linkage stack can be used to assign EAXs to programs and to change the EAX in control register 8 during program linkages. These components are introduced in Linkage-Stack Introduction on page 5-69. The privileged EXTRACT AND SET EXTENDED AUTHORITY instruction also is available for saving and changing the EAX in control register 8.

The SET SECONDARY ASN instruction and the authorization index (AX), bits 32-47 of control register 4, can play a role in the use of access registers. The space-switching form of SET SECONDARY ASN (SSAR-ss) establishes a new secondary address space if the secondary bit selected by the AX is one in the authority table associated with the new secondary space. The secondary space can be addressed by means of an ALET having the value 00000001 hex.

Revoking Accessing Capability: Another mechanism, which is a combined authority and integrity mechanism, is part of access-register translation, and it is described in this section.
An access-list entry (ALE) contains an ASN-second-table-entry sequence number (ASTESN), and so does the ASTE designated by the ALE. During access-register translation, the ASTESN in the ALE must equal the ASTESN in the designated ASTE; otherwise, an exception is recognized.

When the control program allocates an ALE, it should copy the ASTESN from the designated ASTE into the ALE. Subsequently, the control program can, in effect, revoke the addressing capability represented by the ALE by changing the ASTESN in the ASTE. Changing the ASTESN in the ASTE makes all previously usable ALEs that designate the ASTE unusable.

Making an ALE unusable may be required in either of two cases:

1. Some element of the control-program policy for determining the authority of a program to have access to the address space specified by the ASTE has changed. This may mean that some or all of the programs that were authorized to the address space, and for which ALEs have been allocated, are no longer authorized.

Changing the ASTESN in the ASTE ends the usability of all ALEs that designate the ASTE. If this revocation of capability is to be selective, then, when an exception is recognized because of unequal ASTESNs, the control program can reapply its programmed procedures for determining authorization, and an ALE which should have remained usable can be made usable again by copying the new ASTESN into it. When the usability of an ALE is restored, the control program normally should cause reexecution of the instruction that encountered the exception.

2. The ASTE has been reassigned to specify a conceptually different address space, and ALEs which specified the old address space must not be allowed to specify the new one. (Bit 0 of the ASTE, the ASX-invalid bit, can be set to one to delete the assignment of the ASTE to an address space, and this prevents the use of the ASTE in access-register translation. But after reassignment, bit 0 normally is set back to zero.)

The ASTESN mechanism may be regarded as an authority mechanism in the first case above and as an integrity mechanism in the second.

The ASTESN mechanism is especially valuable because it avoids the need of the control program to keep track of the access lists that contain the ALEs that designate each ASTE. Furthermore, it avoids the need of searching through these access lists in order to find the ALEs and set them invalid, to prevent the use of the ALEs in access-register translation. The latter activity could be particularly time-consuming, or could present a particularly difficult management problem, because the access lists could be in auxiliary storage, such as a direct-access storage device, when the need arises to invalidate the ALEs.

The ASTESN is a four-byte field. Assuming a reasonable frequency of authorization-policy changes or address-space reassignments, the approximately four billion possible values of the ASTESN provide a fail-proof authority or integrity mechanism over the lifetime of the system.

Preventing Store References: The access-list entry contains a fetch-only bit which, when one, specifies that the access-list entry cannot be used to perform storage-operand store references. The principal description of the effect of the fetch-only bit is in "Access-List-Controlled Protection" on page 3-11.

Improving Translation Performance: Access-register translation (ART) conceptually occurs each time a logical address is used to reference a storage operand in the access-register mode. To improve performance, ART normally is implemented such that some or all of the information contained in the ART tables (access-list-designation sources, access lists, ASN second tables, and authority tables) is maintained in a special buffer referred to as the ART-lookaside buffer (ALB). The CPU necessarily refers to an ART-table entry in real storage only for the initial access to that entry. The information in the entry may be placed in the ALB, and subsequent translations may be performed using the information in the ALB.

The PURGE ALB instruction and the COMPARE AND SWAP AND PURGE instruction can be used to clear all information from the ALB after a
change has been made to an ART-table entry in real storage.

Access-Register Instructions
The following instructions are provided for examining and changing the contents of access registers:

- COPY ACCESS
- EXTRACT ACCESS
- LOAD ACCESS MULTIPLE
- LOAD ADDRESS EXTENDED
- SET ACCESS
- STORE ACCESS MULTIPLE

The SET ACCESS instruction replaces the contents of a specified access register with the contents of a specified general register. Conversely, the EXTRACT ACCESS instruction moves the contents of an access register to a general register. The COPY ACCESS instruction moves the contents of one access register to another.

The LOAD ACCESS MULTIPLE instruction loads a specified set of consecutively numbered access registers from a specified storage location whose length in words equals the number of access registers loaded. Conversely, the STORE ACCESS MULTIPLE instruction function stores the contents of a set of access registers at a storage location.

The LOAD ADDRESS EXTENDED instruction is similar to the LOAD ADDRESS instruction in that it loads a specified general register with an effective address specified by means of the B, X, and D fields of the instruction. In addition, LOAD ADDRESS EXTENDED operates on the access register having the same number as the general register loaded. When the address-space control, PSW bits 16 and 17, is 00, 10, or 11 binary, LOAD ADDRESS EXTENDED loads the access register with 00000000, 00000001, or 00000002 hex, respectively. When the address space control is 01 binary, LOAD ADDRESS EXTENDED loads the target access register with 00000002 hex when the address-space control is 11 binary is intended to support assignment, by the control program, of entry 2 in the dispatchable-unit access list. Loading the target access register with 00000002 hex when the address-space control is 11 binary is intended to support assignment, by the control program, of entry 2 in the dispatchable-unit access list as specifying the home address space.

Access-Register Translation
Access-register translation is introduced in “Access-Register-Specified Address Spaces” on page 5-44.

Access-Register-Translation Control
Access-register translation is controlled by an address-space control and by controls in control registers 2, 5, and 8. The address-space control, PSW bits 16 and 17, is described in “Translation Modes” on page 3-35. The other controls are described below.

Additional controls are located in the access-register-translation tables.

Control Register 2
The location of the dispatchable-unit control table is specified in control register 2. The register has the following format:

<table>
<thead>
<tr>
<th></th>
<th>DUCTO</th>
<th>58</th>
<th>63</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dispatchable-Unit-Control-Table Origin (DUCTO): Bits 33-57 of control register 2, with six zeros appended on the right, form a 31-bit real
address that designates the beginning of the dispatchable-unit control table. Access-register translation may obtain the dispatchable-unit access-list designation from the dispatchable-unit control table.

Control Register 5
The location of the primary ASN-second-table entry is specified in control register 5. The register has the following format:

```
| 00000000 | PASTE0 |
```

Primary-ASTE Origin (PASTE0): Bits 33-57 of control register 5, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the primary ASN-second-table entry. Access-register translation may obtain the primary-space access-list designation from the primary ASTE. The primary-ASTE origin is set by LOAD ADDRESS SPACE PARAMETERS when it performs PASN translation and by the space-switching forms of PROGRAM CALL, PROGRAM RETURN, and PROGRAM TRANSFER. When any of these instructions places the primary-ASTE origin in control register 5, it also places zeros in bit positions 32 and 58-63 of the register and leaves bits 0-31 of the register unchanged. Bits 0-32 and 58-63 of control register 5 are subject to possible future assignment, and they should not be depended upon to be zeros.

Control Register 8
The extended authorization index is in control register 8. The register has the following format:

```
| 00000000 | PASTE0 |
```

Extended Authorization Index (EAX): Bits 32-47 of control register 8 are the extended authorization index. During access-register translation, the EAX may be compared against the access-list-entry authorization index (ALEAX) in an access-list entry, and it may be used as an index to locate a secondary bit in an authority table. The EAX may be set by a stacking PROGRAM CALL operation, and it is restored by PROGRAM RETURN. The EAX can also be saved and set by the privileged instruction EXTRACT AND SET EXTENDED AUTHORITY.

Access Registers
There are sixteen 32-bit access registers numbered 0-15. The contents of an access register are called an access-list-entry token (ALET). An ALET has the following format:

```
| 00000000 | P | ALESN | ALEN |
```

The fields in the ALET are allocated as follows:

Primary-List Bit (P): When the ALET is not 00000000 or 00000001 hex, bit 7 specifies the access list to be used by access-register translation. When bit 7 is zero, the dispatchable-unit access list is used; this is specified by the dispatchable-unit access-list designation in the dispatchable-unit control table designated by the contents of control register 2. When bit 7 is one, the primary-space access list is used; this is specified by the primary-space access-list designation in the primary ASTE designated by the contents of control register 5.

Access-List-Entry Sequence Number (ALESN): Bits 8-15 may be used as a check on whether the access-list entry designated by the ALET has been invalidated and reallocated since the ALET was obtained. During access-register translation when the ALET is not 00000000 or 00000001 hex, bits 8-15 of the ALET are compared against the access-list-entry sequence number (ALESN) in the designated access-list entry.

Access-List-Entry Number (ALEN): When the ALET is not 00000000 or 00000001 hex, bits 16-31 of the ALET designate an entry in either the dispatchable-unit access list or the primary-space access list, as determined by bit 7. The access-list designation that is used is called the effective access-list designation; it consists of the effective access-list origin and the effective access-list length.

During access-register translation, the ALEN, with four zeros appended on the right, is added to the 31-bit real or absolute address specified by the effective access-list origin, and the result is the real or absolute address of the designated access-list entry. The ALEN is compared against the
effective access-list length to determine whether the designated access-list entry is within the list, and an ALEN-translation exception is recognized if the entry is outside the list. Although the largest possible value of the ALEN is 65,535, an access list can contain at most 1,024 entries.

Bits 0-6 must be zeros during access-register translation; otherwise, an ALET-specification exception is recognized.

When the ALET is 00000000 or 00000001 hex, it specifies the primary or secondary address space, respectively, and the above format does not apply.

Access register 0 usually is treated in access-register translation as containing 00000000 hex, and its actual contents are not examined; the access-register translation done as part of TEST ACCESS is the only exception. Access register 0 is also treated as containing 00000000 hex when it is designated by the B field of LOAD ADDRESS EXTENDED when PSW bits 16 and 17 are 01 binary. When access register 0 is specified for TEST ACCESS or as a source for COPY ACCESS, EXTRACT ACCESS, or STORE ACCESS MULTIPLE, the actual contents of the access register are used. Access register 0, like any other access register, can be loaded by COPY ACCESS, LOAD ACCESS MULTIPLE, LOAD ADDRESS EXTENDED, and SET ACCESS.

Another definition of ALETs 00000000 and 00000001 hex is given in "BRANCH IN SUB-SPACE GROUP" on page 10-13.

**Access-Register-Translation Tables**

When the ALET being translated is not 00000000 or 00000001 hex, access-register translation performs a two-level lookup to locate first the effective access-list designation and then an entry in the effective access list. The effective access-list designation resides in real storage. The effective access list resides in real or absolute storage.

Access-register translation uses an origin in the access-list entry to locate an ASN-second-table entry, and it may perform a one-level lookup to locate an entry in an authority table. The ASN-second-table entry resides in real storage.

The authority table resides in real or absolute storage.

Authority-table entries are described in "Authority-Table Entries" on page 3-31. Access-list designations, access-list entries, and ASN-second-table entries are described in the following sections.

**Dispatchable-Unit Control Table and Access-List Designations**

When the ALET being translated is not 00000000 or 00000001 hex, access-register translation obtains the dispatchable-unit access-list designation if bit 7 of the ALET is zero, or it obtains the primary-space access-list designation if bit 7 is one. The obtained access-list designation is called the effective access-list designation.

The dispatchable-unit access-list designation (DUALD) is located in bytes 16-19 of a 64-byte area called the dispatchable-unit control table (DUCT). The DUCT resides in real storage, and its location is specified by the DUCT origin in control register 2.

The dispatchable-unit control table has the following format:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
</tr>
</tbody>
</table>

In the 24-Bit or 31-Bit Addressing Mode

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td>24</td>
<td>36</td>
</tr>
</tbody>
</table>

Chapter 5. Program Execution 5-53
In the 64-Bit Addressing Mode

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>32</td>
<td>Bits 0-31 of Return Address</td>
</tr>
<tr>
<td>24</td>
<td>36</td>
<td>Bits 32-63 of Return Address</td>
</tr>
</tbody>
</table>

Bytes 0-7 (BASTEO, SA, and SSASTEO) and 12-15 (SSASTESN) of the DUCT are described in "Subspace-Group Dispatchable-Unit Control Table" on page 5-64. Bytes 20-23 (PSW key mask, PSW key, RA, and P) and 32-39 (BA and return address) are described in "BRANCH AND SET AUTHORITY" on page 10-6. Bytes 44-47 (trap-control-block address and E) are described in "TRAP" on page 10-133. Bytes 8-11, 24-27, 40-43, and 48-63 are reserved for possible future extensions and should contain all zeros. Bytes 28-31 are available for use by programming.

The primary-space access-list designation (PSALD) is located in bytes 16-19 of a 64-byte area called the primary ASN-second-table entry. The primary ASTE resides in real storage, and its location is specified by the primary-ASTE origin in control register 5. The format of the primary ASTE is described in "ASN-Second-Table Entries" on page 5-55.

The dispatchable-unit and primary-space access-list designations both have the same format, which is as follows:

Access-List Designation

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Access-List Origin</td>
<td>ALL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

The fields in the access-list designation are allocated as follows:

**Access-List Origin**: Bits 1-24 of the access-list designation, with seven zeros appended on the right, form a 31-bit address that designates the beginning of the access list. This address is treated unpredictably as either a real address or an absolute address.

**Access-List Length (ALL)**: Bits 25-31 of the access-list designation specify the length of the access list in units of 128 bytes, thus making the length of the access list variable in multiples of eight 16-byte entries. The length of the access list, in units of 128 bytes, is one more than the value in bit positions 25-31. The access-list length, with six zeros appended on the left, is compared against bits 0-12 of an access-list-entry number (bits 16-28 of an access-list-entry token) to determine whether the access-list-entry number designates an entry in the access list.

Bit 0 is reserved for a possible future extension and should be zero.

**Programming Note**: The maximum number of access-list entries allowed by an access-list designation is 1,024. There are two access lists available for use at any time. Therefore, a maximum of 2,048 16E-byte address spaces can be addressable without control-program intervention, which is a total of $2^{75}$ bytes.

**Access-List Entries**
The effective access list is the dispatchable-unit access list if bit 7 of the ALET being translated is zero, or it is the primary-space access list if bit 7 is one. The entry fetched from the effective access list is 16 bytes in length and has the following format:

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>O</th>
<th>P</th>
<th>ALESN</th>
<th>ALEAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

The dispatchable-unit and primary-space access-list designations both have the same format, which is as follows:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Access-List Origin</td>
<td>ALL</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>63</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ASTEO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>90</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>
The fields in the access-list entry are allocated as follows:

**ALEN-Invalid Bit (I):** Bit 0, when zero, indicates that the access-list entry specifies an address space. When bit 0 is one during access-register translation, an ALEN-translation exception is recognized.

**Fetch-Only Bit (FO):** Bit 6 controls which types of operand references are permitted to the address space specified by the access-list entry. When bit 6 is zero, both fetch-type and store-type references are permitted. When bit 6 is one, only fetch-type references are permitted, and an attempt to store causes a protection exception for access-list-controlled protection to be recognized and the operation to be suppressed.

**Private Bit (P):** Bit 7, when zero, specifies that any program is authorized to use the access-list entry in access-register translation. When bit 7 is one, authorization is determined as described for bits 16-31.

**Access-List-Entry Sequence Number (ALESN):** Bits 8-15 are compared against the ALESN in the ALET during access-register translation. Inequality causes an ALE-sequence exception to be recognized. It is intended that the control program change bits 8-15 each time it reallocates the access-list entry.

**Access-List-Entry Authorization Index (ALEAX):** Bits 16-31 may be used to determine whether the program for which access-register translation is being performed is authorized to use the access-list entry. The program is authorized if any of the following conditions is met:

1. Bit 7 is zero.
2. Bits 16-31 are equal to the extended authorization index (EAX) in control register 8.
3. The EAX selects a secondary bit that is one in the authority table for the specified address space.

An extended-authority exception is recognized if none of the conditions is met.

**ASN-Second-Table-Entry Origin (ASTEO):** Bits 65-89, with six zeros appended on the right, form the 31-bit real address of the ASTE for the specified address space. Access-register translation obtains the address-space-control element for the address space from the ASTE.

**ASTE Sequence Number (ASTESN):** Bits 96-127 may be used to revoke the addressing capability represented by the access-list entry. Bits 96-127 are compared against an ASTE sequence number (ASTESN) in the designated ASTE during access-register translation.

Bits 1-5, 32-64, and 90-95 are reserved for possible future extensions and should be zeros.

In both the dispatchable-unit access list and the primary-space access list, access-list entries 0 and 1 are intended not to be used in access-register translation. Bits 1-127 of access-list entry 0 and bits 1-63 of access-list entry 1 are reserved for possible future extensions and should be zeros. Bit 0 of access-list entries 0 and 1, and bits 64-127 of access-list entry 1, are available for use by programming. The control program should set bit 0 of access-list entries 0 and 1 to one in order to prevent the use of these entries by means of ALETs in which the ALEN is 0 or 1.

**ASN-Second-Table Entries**

The first 48 bytes of the 64-byte ASN-second-table entry have the following format:

```
<table>
<thead>
<tr>
<th></th>
<th>ATO</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>30 31</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>AX</th>
<th>ATL</th>
<th>CA</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>48</td>
<td>60</td>
<td>63</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ASCE (RTD, STD, or RSD) Part 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTO, STO, or RSTKO</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>
```
The fields in bytes 0-47 of the ASN-second-table entry (ASTE) are defined with respect to certain mechanisms and instructions in "ASN-Second-Table Entries" on page 3-25. The fields in the ASTE are defined with respect to the BRANCH IN SUBSPACE GROUP instruction in "Subspace-Group ASN-Second-Table Entries" on page 5-66. With respect to access-register translation only, and only for an instruction other than BRANCH IN SUBSPACE GROUP, the fields in the ASTE are allocated as follows:

**ASX-Invalid Bit (I):** Bit 0 controls whether the address space associated with the ASTE is available. When bit 0 is zero, access-register translation proceeds. When the bit is one, an ASTE-validity exception is recognized.

**Authority-Table Origin (ATO):** Bits 1-29, with two zeros appended on the right, form a 31-bit address that designates the beginning of the authority table. This address is treated unpredictably as either a real address or an absolute address, although it is treated as a real address for ASN authorization. The authority table is accessed in access-register translation only if the private bit in the access-list entry is one and the access-list-entry authorization index (ALEAX) in the access-list entry is not equal to the extended authorization index (EAX) in control register 8.

**Base-Space Bit (B):** Bit 31 is ignored during access-register translation. Bit 31 is further described in "Subspace-Group ASN-Second-Table Entries" on page 5-66.

**Authorization Index (AX):** Bits 32-47 are not used in access-register translation.

**Authority-Table Length (ATL):** Bits 48-59 specify the length of the authority table in units of four bytes, thus making the authority table variable in multiples of 16 entries. The length of the authority table, in units of four bytes, is one more than the ATL value. The contents of the ATL field are used to establish whether the entry designated by a particular EAX is within the authority table. An extended-authority exception is recognized if the entry is not within the table.
Controlled-ASN Bit (CA): Bit 62 is not used in access-register translation.

Reusable-ASN Bit (RA): Bit 63 is not used in access-register translation.

Address-Space-Control Element (ASCE): Bits 64-127 are an eight-byte address-space-control element (ASCE) that may be a segment-table designation (STD), a region-table designation (RTD), or a real-space designation (RSD). The term “region-table designation” is used to mean a region-first-table designation, region-second-table designation, or region-third-table designation.) The ASCE field is obtained as the result of access-register translation and is used by DAT to translate the logical address for the storage-operand reference being made. Bit 121, the space-switch-event control, is not used in or as a result of access-register translation. The other fields in the ASCE (RTO, STO, RSTKO, G, P, S, R, DT, and TL) are described in “Control Register 1” on page 3-36.

Access-List Designation (ALD): When this ASTE is designated by the primary-ASTE origin in control register 5, bits 128-159 are the primary-space access-list designation (PSALD). See the description of the access-list designation in “Dispatchable-Unit Control Table and Access-List Designations” on page 5-53. During access-register translation when the primary-list bit, bit 7, in the ALET being translated is one, the PSALD is the effective access-list designation.

ASN-Second-Table-Entry Sequence Number (ASTESN): Bits 160-191 are used to control revocation of the accessing capability represented by access-list entries that designate the ASTE. During access-register translation, bits 160-191 are compared against the ASTESN in the access-list entry, and inequality causes an ASTE-sequence exception to be recognized. It is intended that the control program change the value of bits 160-191 when the authorization policies for the address space specified by the ASTE change or when the ASTE is reassigned to specify another address space.

Linkage-Table Designation (LTD) or Linkage-First-Table Designation (LFTD): Bits 192-223 are not used in access-register translation.

ASN-Second-Table-Entry Instance Number (ASTEIN): Bits 352-383 are not used in access-register translation.

Bits 224-319 in the ASTE are available for use by programming.

Programming Note: All unused fields in the ASTE, including the unused fields in bytes 0-31 and all of bytes 32-63, should be set to zeros. These fields are reserved for future extensions, and programs which place nonzero values in these fields may not operate compatibly on future machines.

Access-Register-Translation Process

This section describes the access-register-translation process as it is performed during a storage-operand reference in the access-register mode. LOAD REAL ADDRESS and STORE REAL ADDRESS when PSW bits 16 and 17 are 01 binary, TEST ACCESS in any translation mode, and TEST PROTECTION in the access-register mode, perform access-register translation the same as described here, except that, for LOAD REAL ADDRESS, TEST ACCESS, and TEST PROTECTION, the following exceptions cause a setting of the condition code instead of being treated as program-interruption conditions:

- ALET specification
- ALEN translation
- ALE sequence
- ASTE validity
- ASTE sequence
- Extended authority

BRANCH IN SUBSPACE GROUP performs access-register translation as described in “BRANCH IN SUBSPACE GROUP” on page 10-13.

Access-register translation operates on the access register designated in a storage-operand reference in order to obtain an address-space-control element for use by DAT. When one of access-registers 1-15 is designated, the access-list-entry token (ALET) that is in the access register is used to obtain the address-space-control element. When access register 0 is designated, an ALET having the value 00000000 hex is used, except
that TEST ACCESS uses the actual contents of access register 0.

When the ALET is 00000000 or 00000001 hex, the primary or secondary address-space-control element, respectively, is obtained.

When the ALET is other than 00000000 or 00000001 hex, the leftmost seven bits of the ALET are checked for zeros, the primary-list bit in the ALET and the contents of control register 2 or 5 are used to obtain the effective access-list designation, and the access-list entry number (ALEN) in the ALET is used to select an entry in the effective access list.

The access-list entry is checked for validity and for containing the correct access-list-entry sequence number (ALESN).

The ASN-second-table entry (ASTE) addressed by the access-list entry is checked for validity and for containing the correct ASN-second-table-entry sequence number (ASTESN).

Whether the program is authorized to use the access-list entry is determined through the use of one or more of: (1) the private bit and access-list-entry authorization index (ALEAX) in the access-list entry, (2) the extended authorization index (EAX) in control register 8, and (3) an entry in the authority table addressed by the ASN-second-table entry.

If a store-type reference is to be performed, the fetch-only bit in the access-list entry is checked for being zero.

When no exceptions are recognized, the address-space-control element in the ASN-second-table entry is obtained.

In order to avoid the delay associated with references to real or absolute storage, the information fetched from real or absolute storage normally is also placed in a special buffer, the ART-lookaside buffer (ALB), and subsequent translations involving the same information may be performed by using the contents of the ALB. The operation of the ALB is described in "ART-Lookaside Buffer" on page 5-62.

Whenever access to real or absolute storage is made during access-register translation for the purpose of fetching an entry from an access-list-designation source, access list, ASN second table, or authority table, key-controlled protection does not apply.

The principal features of access-register translation, including the effect of the ALB, are shown in Figure 5-12 on page 5-59.
Access-List Designation

<table>
<thead>
<tr>
<th>ALO</th>
<th>ALL</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ALET in Access Register

<table>
<thead>
<tr>
<th>P</th>
<th>ALESN</th>
<th>ALEN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control Register 1

<table>
<thead>
<tr>
<th>PASCE</th>
</tr>
</thead>
</table>

Control Register 7

<table>
<thead>
<tr>
<th>SASCE</th>
</tr>
</thead>
</table>

Explanation:

1. The appropriate ALD is obtained:
   - When $P$ in the ALET is zero (and the ALET is not zero or one), the DUALD in the DUCT is obtained.
   - When $P$ in the ALET is one, the PSALD in the primary ASTE is obtained.

2. Information, which may include the ALD-source origin, ALET, ALO, and EAX, is used to search the ALB. This information, along with information from the ALE, ASTE, and ATE, may be placed in the ALB.

3. The appropriate ASCE is obtained:
   - When the ALET is zero, the PASCE in CR 1 is obtained.
   - When the ALET is one, the SASCE in CR 7 is obtained.
   - When the ALET is larger than one:
     - If a match exists, the ASCE from the ALB is used.
     - If no match exists, tables from real or absolute storage are fetched. The resulting ASCE from the ASTE is obtained, and entries may be formed in the ALB.

Figure 5-12. Access-Register Translation
Selecting the Access-List-Entry Token
When one of access registers 1-15 is designated, or for the access register designated by the R1 field of TEST ACCESS, access-register translation uses the access-list-entry token (ALET) that is in the access register. When access register 0 is designated, except for TEST ACCESS, an ALET having the value 00000000 hex is used, and the contents of access register 0 are not examined.

Obtaining the Primary or Secondary Address-Space-Control Element
When the ALET being translated is 00000000 hex, the primary address-space-control element in control register 1 is obtained. When the ALET is 00000001 hex, the secondary address-space-control element in control register 7 is obtained. In each of these two cases, access-register translation is completed.

Checking the First Byte of the ALET
When the ALET being translated is other than 00000000 or 00000001 hex, bits 0-6 of the ALET are checked for being all zeros. If bits 0-6 are not all zeros, an ALET-specification exception is recognized, and the operation is suppressed.

Obtaining the Effective Access-List Designation
The primary-list bit, bit 7, in the ALET is used to perform a lookup to obtain the effective access-list designation. When bit 7 is zero, the effective ALD is the dispatchable-unit ALD located in bytes 16-19 of the dispatchable-unit control table (DUCT). When bit 7 is one, the effective ALD is the primary-space ALD located in bytes 16-19 of the primary ASN-second-table entry (primary ASTE).

When bit 7 is zero, the 31-bit real address of the dispatchable-unit ALD is obtained by appending six zeros on the right to the DUCT origin, bits 33-57 of control register 2, and adding 16. The addition cannot cause a carry into bit position 0.

When bit 7 is one, the 31-bit real address of the primary-space ALD is obtained by appending six zeros on the right to the primary-ASTE origin, bits 33-57 of control register 5, and adding 16. The addition cannot cause a carry into bit position 0.

The obtained 31-bit real address is used to fetch the effective ALD — either the dispatchable-unit ALD or the primary-space ALD, depending on bit 7 of the ALET. The fetch of the effective ALD appears to be word concurrent, as observed by other CPUs, and is not subject to protection. When the storage address that is generated for fetching the effective ALD refers to a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed. When the primary-space ALD is fetched, bit 0, the ASX-invalid bit, in the primary ASTE is ignored.

Access-List Lookup
A lookup in the effective access list is performed. The effective access list is the dispatchable-unit access list if bit 7 of the ALET is zero, or it is the primary-space access list if bit 7 is one. The effective access list is treated unpredictably as being in either real or absolute storage.

The access-list-entry-number (ALEN) portion of the ALET is used to select an entry in the effective access list. The 31-bit real or absolute address of the access-list entry is obtained by appending seven zeros on the right to bits 1-24 of the effective ALD and adding the ALEN, with four rightmost and 11 leftmost zeros appended. When a carry into bit position 0 occurs during the addition, an addressing exception may be recognized, or the carry may be ignored, causing the access list to wrap from 231 - 1 to 0. The 31-bit address is formed and used regardless of whether the current PSW specifies the 24-bit, 31-bit, or 64-bit addressing mode.

As part of the access-list-lookup process, the leftmost 13 bits of the ALEN are compared against the effective access-list length, bits 25-31 of the effective ALD, to establish whether the addressed entry is within the access list. For this comparison, the access-list length is extended with six leftmost zeros. If the value formed from the access-list length is less than the value in the 13 leftmost bits of the ALEN, an ALEN-translation exception is recognized, and the operation is nullified.

The 16-byte access-list entry is fetched by using the real or absolute address. The fetch of the entry appears to be word concurrent as observed by other CPUs, with the leftmost word fetched first. The order in which the remaining three
words are fetched is unpredictable. The fetch access is not subject to protection. When the storage address that is generated for fetching the access-list entry refers to a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the access-list entry indicates whether the access-list entry specifies an address space by designating an ASN-second-table entry. This bit is inspected, and, if it is one, an ALEN-translation exception is recognized, and the operation is nullified.

When bit 0 is zero, the access-list-entry sequence number (ALESN) in bit positions 8-15 of the access-list entry is compared against the ALESN in the ALET to determine whether the ALET designates the conceptually correct access-list entry. Inequality causes an ALE-sequence exception to be recognized and the operation to be nullified.

Locating the ASN-Second-Table Entry

The ASN-second-table-entry (ASTE) origin in the access-list entry is used to locate the ASTE. Bits 65-89 of the access-list entry, with six zeros appended on the right, form the 31-bit real address of the ASTE.

The 64-byte ASTE is fetched by using the real address. The fetch of the entry appears to be word concurrent as observed by other CPUs, with the leftmost word fetched first, except that the fetch of the address-space-control element in the entry appears to be doubleword concurrent as observed by other CPUs. The order in which the remaining words, after the first word, and the address-space-control element are fetched is unpredictable. The fetch access is not subject to protection. When the storage address that is generated for fetching the ASTE refers to a location which is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Bit 0 of the ASTE indicates whether the ASTE specifies an address space. This bit is inspected, and, if it is one, an ASTE-validity exception is recognized, and the operation is nullified.

When bit 0 is zero, the ASTE sequence number (ASTESN) in bit positions 160-191 of the ASTE is compared against the ASTESN in bit positions 96-127 of the access-list entry to determine whether the addressing capability represented by the access-list entry has been revoked. Inequality causes an ASTE-sequence exception to be recognized and the operation to be nullified.

Authorizing the Use of the Access-List Entry

The private bit, bit 7, in the access-list entry is used to determine whether the program is authorized to use the access-list entry. The access-list-entry authorization index (ALEAX) in bit positions 16-31 of the access-list entry, the extended authorization index (EAX) in bit positions 32-47 of control register 8, and the authority table designated by the ASTE may also be used.

When the private bit is zero, the program is authorized, and the authorization step of access-register translation is completed.

When the private bit is one but the ALEAX is equal to the EAX, the program is authorized, and the authorization step of access-register translation is completed.

When the private bit is one and the ALEAX is not equal to the EAX, a process called the extended-authorization process is performed. Extended authorization uses the EAX to select an entry in the authority table designated by the ASTE, and it tests the secondary-authority bit in the selected entry for being one. The program is authorized if the tested bit is one.

Extended authorization is the same as the secondary-ASN-authorization process described in "ASN Authorization" on page 3-30 except as follows:

- The authority-table origin is treated as a real or absolute address instead of as a real address.
- The EAX in control register 8 is used instead of the authorization index (AX) in control register 4.
- When the value in bit positions 0-11 of the EAX is greater than the authority-table length (ATL) in the ASTE, an extended-authority exception is recognized instead of a secondary-authority exception. The operation is nullified if the extended-authority exception is recognized.
When the private bit is one, the ALEAX is not equal to the EAX, and the secondary bit in the authority-table entry selected by the EAX is not one, an extended-authority exception is recognized, and the operation is nullified.

Checking for Access-List-Controlled Protection

If a store-type reference is to be performed and the fetch-only bit, bit 6, in the access-list entry is one, a protection exception is recognized, and the operation is suppressed.

Obtaining the Address-Space-Control Element from the ASN-Second-Table Entry

When the ALET being translated is other than 00000000 or 00000001 hex and no exception is recognized in the steps described above, access-register translation obtains the address-space-control element from bit positions 64-127 of the ASTE.

Recognition of Exceptions during Access-Register Translation

The exceptions which can be encountered during the access-register-translation process and their priority are shown in the section “Access Exceptions” in Chapter 6, “Interruptions.”

Programming Note: When updating an access-list entry or ASN-second-table entry, the program should change the entry from invalid to valid (set bit 0 of the entry to zero) as the last step of the updating. This ensures, because the leftmost word is fetched first, that words of a partially updated entry will not be fetched.

ART-Lookaside Buffer

To enhance performance, the access-register-translation (ART) mechanism normally is implemented such that access-list designations and information specified in access lists, ASN second tables, and authority tables are maintained in a special buffer, referred to as the ART-lookaside buffer (ALB). Access-list designations, access-list entries, ASN-second-table entries, and authority-table entries are collectively referred to as ART-table entries. The CPU necessarily refers to an ART-table entry in real or absolute storage only for the initial access to that entry. The information in the entry may be placed in the ALB, and subsequent ART operations may be performed using the information in the ALB. The presence of the ALB affects the ART process to the extent that (1) a modification of an ART-table entry in real or absolute storage does not necessarily have an immediate effect, if any, on the translation, (2) the comparison against the access-list length in an access-list designation that is in storage and used in a translation may be omitted if an ALB access-list entry is used, and (3) the comparison against the authority-table length in an ASN-second-table entry that is in storage and used in a translation may be omitted if an ALB authority-table entry is used. In a multiple-CPU configuration, each CPU has its own ALB.

Entries within the ALB are not explicitly addressable by the program.

Information is not necessarily retained in the ALB under all conditions for which such retention is possible. Furthermore, information in the ALB may be cleared under conditions additional to those for which clearing is mandatory.

ALB Structure

The description of the logical structure of the ALB covers the implementation by all systems operating as defined by z/Architecture. The ALB entries are considered as being of four types: ALB access-list designations (ALB ALDs), ALB access-list entries (ALB ALEs), ALB ASN-second-table entries (ALB ASTEs), and ALB authority-table entries (ALB ATEs). An ALB entry is considered as containing within it both the information obtained from the ART-table entry in real or absolute storage and the attributes used to fetch the ART-table entry from real or absolute storage. There is not an indication in an ALB ALD of whether the ALD-source origin used to select the ALD in real storage was the dispatchable-unit-control-table origin or the primary-ASTE origin.

Note: The following sections describe the conditions under which information may be placed in the ALB, the conditions under which information from the ALB may be used for access-register translation, and how changes to the tables affect the ART process.
Formation of ALB Entries
The formation of ALB entries and the effect of any manipulation of the contents of an ART-table entry in real or absolute storage by the program depend on whether the entry is attached to a particular CPU and on whether the entry is valid.

The attached state of an ART-table entry denotes that the CPU to which the entry is attached can attempt to use the entry for access-register translation. The ART-table entry may be attached to more than one CPU at a time.

An access-list entry or ASN-second-table entry is valid when the invalid bit associated with the entry is zero. Access-list designations and authority-table entries have no invalid bit and are always valid. The primary-space access-list designation is valid regardless of the value of the invalid bit in the primary ASTE.

An ART-table entry may be placed in the ALB whenever the entry is attached and valid.

An access-list designation is attached to a CPU when the designation is within the dispatchable-unit control table designated by the dispatchable-unit-control-table origin in control register 2 or is within the primary ASTE designated by the primary-ASTE origin in control register 5.

An access-list entry is attached to a CPU when the entry is within the access list specified by either an attached access-list designation (ALD) or a usable ALB ALD. A usable ALB ALD is explained in the next section.

An ASN-second-table entry is attached to a CPU when it is designated by the ASTE origin in either an attached and valid access-list entry (ALE) or a usable ALB ALE. A usable ALB ALE is explained in the next section.

An authority-table entry is attached to a CPU when it is within the authority table designated by either an attached and valid ASN-second-table entry (ASTE) or a usable ALB ASTE. A usable ALB ASTE is explained in the next section.

Use of ALB Entries
The usable state of an ALB entry denotes that the CPU can attempt to use the ALB entry for access-register translation. A usable ALB entry attaches the next-lower-level table, if any, and may be usable for a particular instance of access-register translation.

An ALB ALD is in the usable state when the ALDSO field in the ALB ALD matches the current dispatchable-unit-control-table origin or the current primary-ASTE origin.

An ALB ALD may be used for a particular instance of access-register translation when either of the following conditions is met:

1. The primary-list bit in the ALET to be translated is zero, and the ALDSO field in the ALB ALD matches the current dispatchable-unit-control-table origin.

2. The primary-list bit in the ALET to be translated is one, and the ALDSO field in the ALB ALD matches the current primary-ASTE origin.

An ALB ALE is in the usable state when the ALO field in the ALB ALE matches the ALO field in an attached ALD or a usable ALB ALD.

An ALB ALE may be used for a particular instance of access-register translation when all of the following conditions are met:

1. The ALET to be translated has a value larger than 1. (If the ALET is 0 or 1, the contents of CR 1 or CR 7 are used.)

2. The ALO field in the ALB ALE matches the ALO field in the ALD or ALB ALD being used in the translation.

3. The ALEN field in the ALB ALE matches the ALEN field in the ALET to be translated.

An ALB ASTE is in the usable state when the ASTEO field in the ALB ASTE matches the ASTEO field in an attached and valid ALE or a usable ALB ALE.

An ALB ASTE may be used for a particular instance of access-register translation when the ASTEO field in the ALB ASTE matches the ASTEO field in the ALE or ALB ALE being used in the translation.
An ALB ATE may be used for a particular instance of access-register translation when both of the following conditions are met:

1. The ATO field in the ALB ATE matches the ATO field in the ASTE or ALB ASTE being used in the translation.
2. The EAX field in the ALB ATE matches the current EAX.

**Modification of ART Tables**
When an attached but invalid ART-table entry is made valid, or when an unattached but valid ART-table entry is made attached, and no entry formed from the ART-table entry is already in the ALB, the change takes effect no later than the end of the current instruction.

When an attached and valid ART-table entry is changed, and when, before the ALB is cleared of copies of that entry, an attempt is made to perform ART requiring that entry, unpredictable results may occur, to the following extent. The use of the new value may begin between instructions or during the execution of an instruction, including the instruction that caused the change. Moreover, until the ALB is cleared of copies of the entry, the ALB may contain both the old and the new values, and it is unpredictable whether the old or new value is selected for a particular ART operation. If the old and new values are used as representations of effective space designations, failure to recognize that the effective space designations are the same may occur, with the result that operand overlap may not be recognized. Effective space designations and operand overlap are discussed in "Interlocks within a Single Instruction" on page 5-91.

When LOAD ACCESS MULTIPLE or LOAD CONTROL changes the parameters associated with ART, the values of these parameters at the start of the operation are in effect for the duration of the operation.

All entries are cleared from the ALB by the execution of PURGE ALB, a COMPARE AND SWAP AND PURGE instruction that purges the ALB, and SET PREFIX, and by CPU reset.

---

**Subspace Groups**
The subspace-group facility includes the BRANCH IN SUBSPACE GROUP instruction, allocations of fields in the address-space-control element, dispatchable-unit control table, and ASN-second-table entry, and subspace-replacement operations of the PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS instructions. BRANCH IN SUBSPACE GROUP is introduced in "Subroutine Linkage without the Linkage Stack" on page 5-11 and described in detail in "BRANCH IN SUBSPACE GROUP" on page 10-13.

---

**Subspace-Group Tables**
This section describes the use of the dispatchable-unit control table and ASN-second-table entry by the subspace-group facility.

**Subspace-Group Dispatchable-Unit Control Table**
The dispatchable-unit control table has the following format:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>BASTEO</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>SA SSASTEO</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>SSASTESN</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>DUALD</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>PSW-Key Mask PSW Key A P</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
<td>///////////////</td>
</tr>
</tbody>
</table>

---

Subspace Groups: The subspace-group facility includes the BRANCH IN SUBSPACE GROUP instruction, allocations of fields in the address-space-control element, dispatchable-unit control table, and ASN-second-table entry, and subspace-replacement operations of the PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS instructions. BRANCH IN SUBSPACE GROUP is introduced in “Subroutine Linkage without the Linkage Stack” on page 5-11 and described in detail in “BRANCH IN SUBSPACE GROUP” on page 10-13.

Subspace-Group Tables: This section describes the use of the dispatchable-unit control table and ASN-second-table entry by the subspace-group facility.

Subspace-Group Dispatchable-Unit Control Table: The dispatchable-unit control table has the following format:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>BASTEO</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>SA SSASTEO</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>SSASTESN</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>DUALD</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>PSW-Key Mask PSW Key A P</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
<td>///////////////</td>
</tr>
</tbody>
</table>

---

Subspace Groups: The subspace-group facility includes the BRANCH IN SUBSPACE GROUP instruction, allocations of fields in the address-space-control element, dispatchable-unit control table, and ASN-second-table entry, and subspace-replacement operations of the PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS instructions. BRANCH IN SUBSPACE GROUP is introduced in “Subroutine Linkage without the Linkage Stack” on page 5-11 and described in detail in “BRANCH IN SUBSPACE GROUP” on page 10-13.

Subspace-Group Tables: This section describes the use of the dispatchable-unit control table and ASN-second-table entry by the subspace-group facility.

Subspace-Group Dispatchable-Unit Control Table: The dispatchable-unit control table has the following format:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>BASTEO</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>SA SSASTEO</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>SSASTESN</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>DUALD</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>PSW-Key Mask PSW Key A P</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
<td>///////////////</td>
</tr>
</tbody>
</table>
The fields in the dispatchable-unit control table that are used by the subspace-group facility are allocated as follows:

**Base-ASTE Origin (BASTEO):** Bits 1-25 of bytes 0-3, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the ASN-second-table entry that specifies the base space of a subspace group associated with the dispatchable unit. A comparison of bits 1-25 of bytes 0-3 to the primary-ASTE origin (PASTEO) in bit positions 33-57 of control register 5 is made by BRANCH IN SUBSPACE GROUP to determine whether the current primary address space is in the subspace group for the current dispatchable unit. For this comparison, either bits 1-25 may be compared to the PASTEO or the entire contents of bytes 0-3 may be compared to the PASTEO with one leftmost and six rightmost zeros appended. A comparison of bits 1-25 of bytes 0-3 to an ASTE origin (ASTEO) obtained by ASN translation may be made by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS. For this comparison, either bits 1-25 may be compared to the ASTEO or the entire contents of bytes 0-3 may be compared to the ASTEO with one leftmost and six rightmost zeros appended. When BRANCH IN SUBSPACE GROUP uses ALET 0, bits 1-25 of bytes 0-3, with six zeros appended on the right, designate the destination ASTE.

**Subspace-Active Bit (SA):** Bit 0 of bytes 4-7 indicates, when one, that the last BRANCH IN SUBSPACE GROUP instruction executed for the dispatchable unit transferred control to a subspace of the subspace group associated with the dispatchable unit. Bit 0 being zero indicates any one of the following: the last BRANCH IN SUBSPACE GROUP instruction executed for the dispatchable unit transferred control to the base space of the subspace group, BRANCH IN SUBSPACE GROUP has not yet been executed for the dispatchable unit, or the dispatchable unit is not associated with a subspace group. BRANCH IN SUBSPACE GROUP sets bit 0 of bytes 4-7 to one when it transfers control to a subspace of the subspace group associated with the dispatchable unit, and it sets bit 0 to zero when it transfers control to the base space of the subspace group.

**Subspace-ASTE Origin (SSASTEO):** Bits 1-25 of bytes 4-7, with six zeros appended on the right, form a 31-bit real address that designates the beginning of the ASN-second-table entry that specifies the subspace last given control by a BRANCH IN SUBSPACE GROUP instruction executed for the dispatchable unit. BRANCH IN SUBSPACE GROUP transfers control to a subspace of the subspace group associated with the dispatchable unit, and it sets bit 0 to zero when it transfers control to the base space of the subspace group.
the subspace-active bit to one and either sets bits 26-31 of bytes 4-7 to zeros or leaves those bits unchanged. However, if bits 1-25 are all zeros, a special-operation exception is recognized. When BRANCH IN SUBSPACE GROUP transfers control to the base space of the subspace group, it sets the subspace-active bit to zero, and bits 1-31 of bytes 4-7 remain unchanged. Bits 1-25 of bytes 4-7 may be used by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS to set bits 0-55 and 57-63 of the primary ASCE in control register 1 or the secondary ASCE in control register 7 from the same bits of the ASCE in the subspace ASTE.

Subspace-ASTE Sequence Number (SSASTESTSN): Bytes 12-15 may be used to revoke the linkage capability represented by the SSASTEO, bits 1-25 of bytes 4-7, in the DUCT. When BRANCH IN SUBSPACE GROUP transfers control to a subspace by means of an ALET other than ALET 1, it obtains the ASTESN in the subspace ASTE and places it in bytes 12-15. When BRANCH IN SUBSPACE GROUP uses ALET 1 to transfer control to a subspace, it compares bytes 12-15 to the ASTESN in the subspace ASTE, and it recognizes an ASTE-sequence exception if they are unequal. When the SSASTEO is used by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and LOAD ADDRESS SPACE PARAMETERS to set bits 0-55 and 57-63 of the primary ASCE in control register 1 or the secondary ASCE in control register 7 from the same bits of the ASCE in the subspace ASTE, those instructions first compare bytes 12-15 to the ASTESN in the subspace ASTE, and they recognize an ASTE-sequence exception if the two fields are unequal.

Bytes 16-19 are described in "Dispatchable-Unit Control Table and Access-List Designations" on page 5-53. Bytes 20-23 are described in "BRANCH AND SET AUTHORITY" on page 10-6. Bytes 32-39 and 44-47 are described in "TRAP" on page 10-133. Bytes 24-27, 40-43, and 48-63 are reserved for possible future extensions and should contain all zeros. Bytes 28-31 are available for use by programming.

Subspace-Group ASN-Second-Table Entries
The 64-byte ASN-second-table entries have the following format:

<table>
<thead>
<tr>
<th>I</th>
<th>ATO</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>30 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AX</th>
<th>ATL</th>
<th>C R A A</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>48</td>
<td>60 63</td>
</tr>
</tbody>
</table>

ASCE (RTD, STD, or RSD) Part 1

<table>
<thead>
<tr>
<th>RTO, STO, or RSTKO</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>95</td>
</tr>
</tbody>
</table>

RTD or STD Part 2

<table>
<thead>
<tr>
<th>RTO/STO (Cont.)</th>
<th>GPSX</th>
<th>R</th>
<th>DT</th>
<th>TL</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>115</td>
<td>118 122 124 127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RSD Part 2

<table>
<thead>
<tr>
<th>RSTKO (Cont.)</th>
<th>GPSX</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>115</td>
<td>118 122 127</td>
</tr>
</tbody>
</table>

ALD

<table>
<thead>
<tr>
<th>ALO</th>
<th>ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>153 159</td>
</tr>
</tbody>
</table>

ASTESN

| 160 191 |

If ASN-and-LX Reuse Is Not Enabled

<table>
<thead>
<tr>
<th>LTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>192 217 223</td>
</tr>
</tbody>
</table>
If ASN-and-LX Reuse Is Enabled

<table>
<thead>
<tr>
<th>V</th>
<th>LFTO</th>
<th>LFTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>192</td>
<td>216</td>
<td>223</td>
</tr>
</tbody>
</table>

Available for programming

<table>
<thead>
<tr>
<th>224</th>
<th>255</th>
</tr>
</thead>
</table>

Available for programming

<table>
<thead>
<tr>
<th>256</th>
<th>287</th>
</tr>
</thead>
</table>

Available for programming

<table>
<thead>
<tr>
<th>288</th>
<th>319</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>320</th>
<th>351</th>
</tr>
</thead>
</table>

ASTEIN

<table>
<thead>
<tr>
<th>352</th>
<th>383</th>
</tr>
</thead>
</table>

For BRANCH IN SUBSPACE GROUP, the fields in bytes 0-47 of the ASTE are allocated as follows:

**ASX-Invalid Bit (I):** Bit 0 controls whether the address space associated with the ASTE is available. When bit 0 is zero during access-register translation of ALET 1 or an ALET other than 0 and 1 for BRANCH IN SUBSPACE GROUP, the translation proceeds. When the bit is one, an ASTE-validity exception is recognized. The bit is ignored during access-register translation of ALET 0. When the ASTE is designated by a subspace-ASTE origin (SSASTEO) in a dispatchable-unit control table, bit 0 is also used as described in the definition of bits 160-191 (ASTESN).

**Authority-Table Origin (ATO):** Bits 1-29 are not used by BRANCH IN SUBSPACE GROUP.

**Base-Space Bit (B):** Bit 31 specifies, when one, that the address space associated with the ASTE is the base space of a subspace group. When BRANCH IN SUBSPACE GROUP uses an ALET other than ALETs 0 and 1 to locate a destination ASTE, it recognizes a special-operation exception if the destination-ASTE origin does not equal the base-ASTE origin in the dispatchable-unit control table and bit 31 is one in the destination ASTE.

**Authorization Index (AX):** Bits 32-47 are not used by BRANCH IN SUBSPACE GROUP.

**Authority-Table Length (ATL):** Bits 48-59 are not used by BRANCH IN SUBSPACE GROUP.

**Controlled-ASN Bit (CA):** Bit 62 is not used by BRANCH IN SUBSPACE GROUP.

**Reusable-ASN Bit (RA):** Bit 63 is not used by BRANCH IN SUBSPACE GROUP.

**Address-Space-Control Element (ASCE):** Bits 64-127 are an eight-byte address-space-control element (ASCE) that may be a segment-table designation (STD), a region-table designation (RTD), or a real-space designation (RSD). (The term “region-table designation” is used to mean a region-first-table designation, region-second-table designation, or region-third-table designation.) The ASCE field is obtained as the result of access-register translation done for BRANCH IN SUBSPACE GROUP. When BRANCH IN SUBSPACE GROUP uses an ALET other than ALETs 0 and 1 to locate a destination ASTE, it recognizes a special-operation exception if the destination-ASTE origin does not equal the base-ASTE origin in the dispatchable-unit control table and the subspace-group-control bit, bit 118 (G), in the destination ASTE is zero. When BRANCH IN SUBSPACE GROUP transfers control to the base space of a subspace group associated with the current dispatchable unit, it places bits 64-127 in control register 1; otherwise, when BRANCH IN SUBSPACE GROUP transfers control to a subspace of the subspace group, it places bits 64-119 and 121-127 in bit positions 0-55 and 57-63, respectively, of control register 1. Bits 64-127 are used after ASN translation by PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, SET SECONDARY ASN, and LOAD ADDRESS SPACE PARAMETERS as described in "ASN-Second-Table Entries” on page 3-25.
**Access-List Designation (ALD):** When this ASTE is designated by the primary-ASTE origin in control register 5, bits 128-159 are the primary-space access-list designation (PSALD). During access-register translation when the primary-list bit, bit 7, in the ALET being translated is one, the PSALD is the effective access-list designation.

**ASN-Second-Table-Entry Sequence Number (ASTESN):** Bits 160-191 are used to control revocation of the accessing capability represented by access-list entries that designate the ASTE. During access-register translation, bits 160-191 are compared against the ASTESN in the access-list entry, and inequality causes an ASTE-sequence exception to be recognized.

**Linkage-Table Designation (LTD) or Linkage-First-Table Designation (LFTD):** Bits 192-223 are not used by BRANCH IN SUBSPACE GROUP.

**ASN-Second-Table-Entry Instance Number (ASTEIN):** Bits 352-383 are not used by BRANCH IN SUBSPACE GROUP.

Bits 224-319 in the ASTE are available for use by programming.

When the ASTE is designated by a subspace-ASTE origin (SSASTEO) in a dispatchable-unit control table, bits 160-191 are also used to control revocation of the linkage capability represented by that SSASTEO. When BRANCH IN SUBSPACE GROUP uses ALET 1 to transfer control to the subspace specified by the SSASTEO, or when PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, SET SECONDARY ASN, or LOAD ADDRESS SPACE PARAMETERS uses the SSASTEO to set bits 0-55 and 57-63 of the primary ASCE in control register 1 or the secondary ASCE in control register 7 from the same bits of the ASCE in the subspace ASTE, those instructions first test bit 0 of the subspace ASTE for being zero and recognize an ASTE-validity exception if it is not, and they then compare bits 160-191 to the subspace-ASTE sequence number (SSASTESN) in the dispatchable-unit control table and recognize an ASTE-sequence exception if there is an inequality. However, when either of the two named exception conditions exists for LOAD ADDRESS SPACE PARAMETERS, the instruction sets condition code 1 or 2 instead of recognizing the exception.

**Programming Note:** All unused fields in the ASTE, including the unused fields in bytes 0-31 and all of bytes 32-63, should be set to zeros. These fields are reserved for future extensions, and programs which place nonzero values in these fields may not operate compatibly on future machines.

**Subspace-Replacement Operations**

The subspace-group facility includes subspace-replacement operations of PROGRAM CALL, PROGRAM TRANSFER, PROGRAM RETURN, SET SECONDARY ASN, and LOAD ADDRESS SPACE PARAMETERS. The operations apply when the dispatchable unit for which any of the five named instructions is executed is in a state called subspace active. A dispatchable unit is subspace active if it has used BRANCH IN SUBSPACE GROUP to transfer control to a subspace of its subspace group and has not subsequently used BRANCH IN SUBSPACE GROUP to return control to the base space of the group.

The definitions of the subspace-replacement operations are included in the definitions of the five named instructions in Chapter 10, “Control Instructions.” The operations are described in a general way as follows. Whenever (1) an address space is established as the primary or secondary address space as a result of ASN translation or (2) PROGRAM CALL obtains the origin of the ASN-second-table entry specifying a new primary address space from the entry-table entry used, then, if that address space is in a subspace group, as indicated by the subspace-group-control bit, bit 54 (G), being one in the address-space-control element (ASCE) for the address space (the new PASCE in control register 1 or SASCE in control register 7), and if the dispatchable unit is subspace-active, as indicated by the subspace-active bit, bit 0 (SA) of word 1, in the dispatchable-unit control table (DUCT) being one, the ASN-second-table-entry (ASTE) origin (ASTEO) for the address space, which was obtained by ASN translation or from the entry-table entry, is compared to the base-ASTE origin (BASTEO), bits 1-25 of word 0, in the DUCT. If that ASTEO and the BASTEO are equal, the fol-
lowing occurs. An ASTE-validity exception is recognized if bit 0 in the ASTE for the last subspace entered by the dispatchable unit, which ASTE is designated by the subspace-ASTE origin (SSASTEO) in the DUCT, is one. An ASTE-sequence exception is recognized if the ASTE-sequence number (ASTESN) in word 5 of the subspace ASTE does not equal the subspace ASTESN (SSASTESN) in word 3 of the DUCT. However, LOAD ADDRESS SPACE PARAMETERS sets a nonzero condition code instead of recognizing the ASTE-validity or ASTE-sequence exception. If no exception exists, bits 0-55 and 57-63 of the ASCE for the address space (the PASCE in control register 1 or SASCE in control register 7) are replaced by the same bits of the ASCE in word 2 of the subspace ASTE.

If an addressing exception is recognized when attempting to access the DUCT or subspace ASTE, the instruction execution is suppressed. If an ASTE-validity or ASTE-sequence exception is recognized, the instruction execution is nullified. Such nullification or suppression causes all control register contents to remain unchanged from what they were at the beginning of the instruction execution.

Key-controlled protection does not apply to any accesses to the DUCT or subspace ASTE.

For comparing the ASTEO obtained by ASN translation to the BASTEO, either the ASTEO may be compared to the BASTEO or the ASTEO, with one leftmost and six rightmost zeros appended, may be compared to the entire contents of word 0 of the DUCT.

When the SSASTEO in the DUCT is used to access the subspace ASTE, no check is made for whether the SSASTEO is all zeros.

The references to the DUCT and subspace ASTE are single-access references and appear to be word concurrent as observed by other CPUs. The words of the DUCT are accessed in no particular order. The words of the subspace ASTE are accessed in no particular order except that word 0 is accessed first.

The exceptions that can be recognized during a subspace-replacement operation are referred to collectively as the subspace-replacement exceptions and are listed in priority order in "Subspace-Replacement Exceptions" on page 6-47.

Linkage-Stack Introduction

Many of the functions related to the linkage stack are described in this section and in "Linkage-Stack Operations" on page 5-75. Additionally, tracing of the stacking PROGRAM CALL instruction and of the PROGRAM RETURN instruction is described in Chapter 5, "Program Execution"; interruptions in Chapter 6, "Interruptions" and the instructions are described in Chapter 10, "Control Instructions."

Summary

These major functions are provided:

1. A table-based subroutine-linkage mechanism that provides PSW and control-register status changing and which saves and restores this status and the contents of general registers and access registers through the use of an entry in a linkage stack.

2. A branch-type linkage mechanism that uses the linkage stack.

3. Instructions for placing an additional two words of status in the current linkage-stack entry and for retrieving all of the status and the general-register and access-register contents that are in the entry.

4. An instruction for determining whether a program is authorized to use a particular access-list-entry token.

5. Aids for program-problem analysis.

In addition, control and authority mechanisms are incorporated to control these functions.

It is intended that a separate linkage stack be associated with and used by each dispatchable unit. The linkage stack for a dispatchable unit resides in the home address space of the dispatchable unit.

It is intended that a dispatchable unit's linkage stack be protected from the dispatchable unit by means of key-controlled protection. Key-controlled protection does not apply to the linkage-stack instructions that place information in or retrieve information from the linkage stack.
The linkage-stack functions are for use by programs considered to be semiprivileged, that is, programs which are executed in the problem state but which are authorized to use additional functions. With these authorization controls, a nonhierarchical organization of programs may be established, with each program in a sequence of calling and called programs having a degree of authority that is arbitrarily different from those of programs before or after it in the sequence. The range of functions available to each program, and the ability to transfer control from one program to another, are prescribed in tables that are managed by the control program.

The linkage-stack instructions, which are semiprivileged, are described in Chapter 10, “Control Instructions.” They are:

- **BRANCH AND STACK**
- **EXTRACT STACKED REGISTERS**
- **EXTRACT STACKED STATE**
- **MODIFY STACKED STATE**
- **PROGRAM RETURN**
- **TEST ACCESS**

In addition, the PROGRAM CALL instruction optionally forms an entry in the linkage stack. A PROGRAM CALL instruction that operates on the linkage stack is called stacking PROGRAM CALL. Recognition of PROGRAM CALL as stacking PROGRAM CALL is under the control of a bit in the entry-table entry.

### Linkage-Stack Functions

#### Transferring Program Control

The use of the linkage stack permits programs operating at arbitrarily different levels of authority to be linked directly without the intervention of the control program. The degree of authority of each program in a sequence of calling and called programs may be arbitrarily different, thus allowing a nonhierarchical organization of programs to be established. Modular authorization control can be obtained principally by associating an extended authorization index with each program module. This allows program modules with different authorities to coexist in the same address space. On the other hand, the extended authorization index in effect during the execution of a called program module can be the one that is associated with the calling program module, thus allowing the called module to be executed with different authorities on behalf of different dispatchable units. Options concerning the PSW-key mask and the secondary ASN are other means of associating different authorities with different programs or with the same called program. The authority of each program is prescribed in tables that are managed by the control program. By setting up the tables so that the same program can be called by means of different PC numbers, the program can be assigned different authorities depending on which PC number is used to call it. The tables also allow control over which PC numbers can be used by a program to call other programs.

The stacking PROGRAM CALL and PROGRAM RETURN linkage operations can link programs residing in different address spaces and having different levels of authority. The execution state and the contents of the general registers and access registers are saved during the execution of stacking PROGRAM CALL and are partially restored during the execution of PROGRAM RETURN. A linkage stack provides an efficient means of saving and restoring both the execution state and the contents of registers during linkage operations.

During the execution of a PROGRAM CALL instruction, the PC-number-translation process is performed to locate a 32-byte entry-table entry. When the PC-type bit in the entry-table entry is one, the stacking PROGRAM CALL operation is specified; otherwise, the basic PROGRAM CALL operation is specified.

In addition to the information applying to both basic PROGRAM CALL and stacking PROGRAM CALL (described in "PC-Number Translation" on page 5-31 and consisting of an authorization key mask and specifications of the new ASN, addressing mode, instruction address, problem state, PSW-key mask, primary-ASTE address, and entry parameter), the entry-table entry contains information that specifies options concerning the address-space control and PSW key in the PSW, and the PSW-key mask, extended authorization index, and secondary ASN in the control registers.

During the stacking PROGRAM CALL operation and by means of the additional information in the entry-table entry, the address-space control in the PSW can be set to specify either the primary-space mode or the access-register mode. The PSW key can be either left unchanged or replaced...
from the entry-table entry. The PSW-key mask in control register 3 can be either ORed to or replaced from the entry-table entry. The extended authorization index in control register 8 can be either left unchanged or replaced from the entry-table entry. The secondary ASN in control register 3 can be set equal to the primary ASN of either the calling program or the called program; thus, the ability of the called program to have access to the primary address space of the calling program can be controlled.

The stacking PROGRAM CALL operation always forms an entry, called a state entry, in the linkage stack to save the execution state and the contents of general registers 0-15 and access registers 0-15. The saved execution state includes a called-space identification, the numeric part of the PC number used, the updated PSW before any changes are made due to the entry-table entry, the extended authorization index, PSW-key mask, primary ASN, and secondary ASN existing before the operation, and the extended-addressing-mode bit existing after the operation. However, the value of the PER mask in the saved updated PSW is unpredictable. The linkage-stack state entry also contains an entry-type code that identifies the entry as one that was formed by PROGRAM CALL. If the ASN-and-LX-reuse facility is installed and enabled, the state entry also contains the primary ASN-second-table-entry instance number (primary ASTEIN) and secondary ASTEIN existing before the operation.

A space-switching operation occurs when the address-space number (ASN) specified in the entry-table entry is nonzero. When space switching occurs, the operation is called PROGRAM CALL with space switching (PC-ss), and the ASN in the entry-table entry is placed in control register 4 as a new primary ASN. When no space switching occurs, the operation is called PROGRAM CALL to current primary (PC-cp), and there is no change to the primary ASN in control register 4.

PROGRAM CALL with space switching obtains the new ASN from the entry-table entry and places it in control register 4 as the primary ASN. It obtains a new primary-ASTE origin from the entry-table entry and new primary address-space-control element from the new primary ASTE, and it places them in control registers 5 and 1, respectively. It sets the secondary address-space-control element in control register 7 equal to either the old primary address-space-control element, or the new one, depending on whether it sets the secondary ASN in control register 3 equal to the old primary ASN or the new one, respectively. PROGRAM CALL to current primary sets the secondary ASN equal to the primary ASN and the secondary address-space-control element equal to the primary address-space-control element.

If the ASN-and-LX-reuse facility is installed and enabled, PROGRAM CALL with space switching obtains the ASN-second-table-entry instance number (ASTEIN) from the new primary ASTE and places it in control register 4 as the new primary ASTEIN. It sets the secondary ASTEIN in control register 3 equal to either the old primary ASTEIN or the new one, depending on whether it set the secondary ASN equal to the old primary ASN or the new one, respectively. PROGRAM CALL to current primary sets the secondary ASTEIN equal to the primary ASTEIN.

The instruction PROGRAM RETURN restores most of the information saved in the linkage stack by the stacking PROGRAM CALL operation. It restores the PSW, extended authorization index, PSW-key mask, primary ASN, secondary ASN, and the contents of general registers 2-14 and access-registers 2-14. However, the PER mask in the current PSW remains unchanged. If the ASN-and-LX-reuse facility is installed and enabled, PROGRAM RETURN restores the primary ASTEIN and secondary ASTEIN. The operation of PROGRAM RETURN is referred to by saying that PROGRAM RETURN unstacks a state entry.

For PROGRAM RETURN, a space-switching operation occurs when the restored primary ASN is not equal to the primary ASN existing before the operation. When space switching occurs, the operation is called PROGRAM RETURN with space switching (PR-ss). When no space switching occurs, the operation is called PROGRAM RETURN to current primary (PR-cp).

PROGRAM RETURN with space switching performs ASN translation of the restored primary ASN to obtain a new primary-ASTE origin and a new primary address-space-control element, which it places in control registers 5 and 1, respectively. For PROGRAM RETURN with space switching or to current primary, (1) if the restored secondary
ASN is the same as the restored primary ASN, the secondary address-space-control element in control register 7 is set equal to the new primary address-space-control element in control register 1, or (2) if the restored secondary ASN is not the same as the restored primary ASN, ASN translation and ASN authorization of the restored secondary ASN are performed to obtain a new secondary address-space-control element, which is placed in control register 7.

If the ASN-and-LX-reuse facility is installed and enabled, PROGRAM RETURN with space switching requires that the ASN-second-table-entry instance number (ASTEIN) in the new primary ASTE be equal to the primary ASTEIN saved in the state entry being unstacked. PROGRAM RETURN with space switching or to current primary, if the restored secondary ASN is not the same as the restored primary ASN, requires that the ASTEIN in the new secondary ASTE be equal to the saved ASTEIN. An exception is recognized if either of these requirements is not met.

The stacking PROGRAM CALL operation and the PROGRAM RETURN operation each can be performed successfully only in the primary-space mode or access-register mode. An exception is recognized when the CPU is in the real mode, secondary-space mode, or home-space mode.

A bit, named the unstack-suppression bit, can be set to one in a linkage-stack state entry to cause an exception if an attempt is made by PROGRAM RETURN to unstack the entry. When the bit is one, the entry can be operated on by the instructions that add information to or retrieve information from the entry. The unstack-suppression bit is intended to allow the control program to gain control when an attempt is made to unstack a state entry in which the bit is one.

**Branching Using the Linkage Stack**

The execution state and the contents of the general registers and access registers can also be saved in the linkage stack by means of the instruction BRANCH AND STACK. BRANCH AND STACK uses a branch address as do the other branching instructions, instead of using a PC number. BRANCH AND STACK, along with PROGRAM RETURN, can link programs residing in the same address space and having the same level of authority; that is, BRANCH AND STACK does not change the execution state except for the instruction address.

BRANCH AND STACK forms a linkage-stack state entry that is almost the same as one formed by PROGRAM CALL. When it is necessary to distinguish between these two types of state entry, an entry formed by PROGRAM CALL is called a program-call state entry, and one formed by BRANCH AND STACK is called a branch state entry. A branch state entry differs from a program-call state entry in two ways: (1) it contains a different entry-type code, which identifies it as a branch state entry, and (2) it contains the basic-addressing-mode bit and instruction address existing after the operation instead of a called-space identification and the numeric part of the PC number used. These new values of PSW bits 32 and 64-127 are in addition to the complete PSW that is saved in the state entry.

For BRANCH AND STACK, the basic- and extended addressing mode bits and the instruction address that are part of the complete PSW saved in the state entry can be the current (at the beginning of the operation) addressing-mode bits and the updated instruction address (the address of the next sequential instruction), or they can be specified in a register. This register can be one that had link information placed in it by a BRANCH AND LINK (BALR only), BRANCH AND SAVE, BRANCH AND SAVE AND SET MODE, or BRANCH AND SET MODE instruction. Thus, BRANCH AND STACK can be used either in a calling program or at (or near) the entry point of a called program, and, in either case, a PROGRAM RETURN instruction located at the end of the called program will return correctly to the calling program. The ability to use BRANCH AND STACK at an entry point allows the linkage stack to be used without changing old calling programs.

When the R field of BRANCH AND STACK is zero, the instruction is executed without causing branching.

When PROGRAM RETURN unstacks a branch state entry, it ignores the extended authorization index, PSW-key mask, primary ASN, secondary ASN, primary ASTEIN, and secondary ASTEIN in the entry. The PROGRAM RETURN instruction restores the PSW and the contents of general registers 2-14 and access registers 2-14 that were
saved in the entry. However, the PER mask in the current PSW remains unchanged.

BRANCH AND STACK can be executed successfully only in the primary-space mode or access-register mode. An exception is recognized when the CPU is in the real mode, secondary-space mode, or home-space mode.

The unstack-suppression bit has the same effect in a branch state entry as it does in a program-call state entry.

Adding and Retrieving Information
The instruction MODIFY STACKED STATE can be used by a program to place two words of information, contained in a designated general-register pair, in an area, called the modifiable area, of the current linkage-stack state entry (a branch state entry or a program-call state entry). This is intended to allow a called program to establish a recovery routine that will be given control by the control program, if necessary.

The instructions EXTRACT STACKED REGISTERS and EXTRACT STACKED STATE can be used by a program to obtain any of the information saved in the current state entry by BRANCH AND STACK or PROGRAM CALL or placed there by MODIFY STACKED STATE. EXTRACT STACKED REGISTERS (EREGG) places the contents of a specified range of general registers and access registers back in the registers from which the contents were saved. EXTRACT STACKED REGISTERS (EREG) does the same except that it restores only bits 32-63 of the general registers and leaves bits 0-31 unchanged. EXTRACT STACKED STATE obtains pairs of words of the nonregister information saved or placed in a state entry and places them in bit positions 32-63 of a designated general-register pair. Alternatively, EXTRACT STACKED STATE obtains two doublewords containing a PSW saved in the state entry and places them in bit positions 0-63 of a designated general-register pair. If the ASN-and-LX-reuse facility is installed, EXTRACT STACKED STATE can place the contents of two words of the state entry in bit positions 0-31 of a designated general-register pair. These contents are the saved secondary ASTEIN and primary ASTEIN if the ASN-and-LX-reuse facility is enabled. EXTRACT STACKED STATE sets the condition code to indicate whether the current state entry is a branch state entry or a program-call state entry.

Testing Authorization
The instruction TEST ACCESS has as operands an access-list-entry token (ALET) in a designated access register and an extended authorization index (EAX) in a designated general register. TEST ACCESS applies the access-register-translation process, which uses the specified EAX instead of the current EAX in control register 8, to the ALET, and it sets the condition code to indicate the result. The condition code may indicate: (1) the ALET is 00000000 hex, (2) the ALET designates an entry in the dispatchable-unit access list and can be translated without exceptions in access-register translation, (3) the ALET designates an entry in the primary-space access list and can be translated without exceptions in access-register translation, or (4) the ALET is 00000001 hex or causes exceptions in access-register translation.

The principal purpose of TEST ACCESS is to allow a called program to determine whether an ALET passed to it by the calling program is authorized for use by the calling program by means of the calling program's EAX. This is in support of a possible programming convention in which a called program will not operate on an AR-specified address space by means of its own EAX unless the calling program is authorized to operate on that space by means of the calling program's EAX. The called program can obtain the calling program's EAX, for use by TEST ACCESS, from the current linkage-stack state entry by means of the EXTRACT STACKED STATE instruction.

Another purpose of TEST ACCESS is to indicate the special cases in which the ALET is 00000000 hex, designating the primary address space, or 00000001 hex, designating the secondary address space. Because PROGRAM CALL may change the primary and secondary address spaces, ALETs 00000000 hex and 00000001 hex may designate different address spaces when used by the called program than when used by the calling program.

Still another purpose of TEST ACCESS is to indicate whether the ALET designates an entry in the primary-space access list since such a designation after the primary address space was changed by
Program-Problem Analysis

To aid program-problem analysis, the option is provided of having a trace entry made implicitly for three additional linkage operations when the linkage stack is used. When branch tracing is on, a trace entry is made each time a BRANCH AND STACK instruction is executed and causes branching. When ASN tracing is on, a trace entry is made each time the stacking PROGRAM CALL operation is performed and each time PROGRAM RETURN unstacks a linkage-stack state entry formed by PROGRAM CALL. When mode tracing is on, a trace entry for mode tracing is not made if one due to ASN tracing is made. A detailed definition of tracing is contained in "Tracing" on page 4-10.

As a further analysis aid, BRANCH AND STACK when it causes branching, stacking PROGRAM CALL, and PROGRAM RETURN are also recognized as PER successful-branching events. For PROGRAM RETURN, the unstacked state entry may have been formed by BRANCH AND STACK or PROGRAM CALL.

The execution of a space-switching stacking PROGRAM CALL or PROGRAM RETURN instruction causes a space-switch event if the primary space-switch-event control is one before or after the operation or if a PER event is to be indicated.

Linkage-Stack Entry-Table Entries

All of the fields in the entry-table entry except bits 130-159 are described in "Entry-Table Entries" on page 5-34. This section describes only bits 130-159.

The entry-table entry has the following format:

If Bit 129 is Zero

<table>
<thead>
<tr>
<th>A</th>
<th>Entry Instruction Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

If Bit 129 is One

<table>
<thead>
<tr>
<th></th>
<th>Entry Instruction Address (Part 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Entry Instruction Address (Part 2)</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authorization Key Mask</th>
<th>ASN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry Key Mask</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>G</th>
<th>K</th>
<th>C</th>
<th>E</th>
<th>Entry Ext. Auth. Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128 131 136 140 144 159</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASTE Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry Parameter (Part 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry Parameter (Part 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The fields in bit positions 130-159 are allocated as follows:

PSW-Key Control (K): Bit 131, when one, specifies that bits 136-139 are to replace the PSW key in the PSW as part of the stacking PROGRAM CALL operation. When this bit is zero, the PSW key remains unchanged. Bit 131 is ignored during the basic PROGRAM CALL operation.

PSW-Key-Mask Control (M): Bit 132, when one, specifies that bits 96-111 are to replace the PSW-key mask in control register 3 as part of the stacking PROGRAM CALL operation. When this bit is zero, bits 96-111 are ORed into the PSW-key mask in control register 3 as part of the stacking PROGRAM CALL operation. Bit 132 is ignored during the basic PROGRAM CALL operation.
Extended-Authorization-Index Control (E): Bit 133, when one, specifies that bits 144-159 are to replace the current extended authorization index in control register 8 as part of the stacking PROGRAM CALL operation. When this bit is zero, the current extended authorization index remains unchanged. Bit 133 is ignored during the basic PROGRAM CALL operation.

Address-Space-Control Control (C): Bit 134, when one, specifies that bit 17 of the current PSW is to be set to one as part of the stacking PROGRAM CALL operation. When this bit is zero, bit 17 is set to zero. Because the CPU must be in either the primary-space mode or the access-register mode when a stacking PROGRAM CALL instruction is issued, the result is that the CPU is placed in the access-register mode if bit 134 is one or the primary-space mode if bit 134 is zero. Bit 134 is ignored during the basic PROGRAM CALL operation.

Secondary-ASN Control (S): Bit 135, when one, specifies that bits 80-95 are to become the new secondary ASN, and the new SASCE is to be set equal to the new PASCE, as part of the stacking PROGRAM CALL with-space-switching (PC-ss) operation. When this bit is zero, the new SASN and SASCE are set equal to the PASN and PASCE, respectively, of the calling program. When the ASN-and-LX-reuse facility is installed and enabled, bit 135 similarly specifies, when one, that the new SASTEIN is to be set equal to the new PASTEIN or, when zero, that the new SASTEIN is to be set equal to the PASTEIN of the calling program. Bit 135 is ignored during the basic PROGRAM CALL operation and the stacking PROGRAM CALL to-current-primary (PC-cp) operation.

Entry Key (EK): Bits 136-139 replace the PSW key in the PSW as part of the stacking PROGRAM CALL operation if the PSW-key control, bit 131, is one. Bits 136-139 are ignored, and the current PSW key remains unchanged, if bit 131 is zero. Bits 136-139 are ignored during the basic PROGRAM CALL operation.

Entry Extended Authorization Index: Bits 144-159 replace the current extended authorization index, bits 32-47 of control register 8, as part of the stacking PROGRAM CALL operation if the extended-authorization-index control, bit 133, is one. Bits 144-159 are ignored, and the current extended authorization index remains unchanged, if bit 133 is zero. Bits 144-159 are ignored during the basic PROGRAM CALL operation.

Bits 130 and 140-143 are reserved for possible future extensions and should be zeros.

Linkage-Stack Operations

A linkage stack may be formed by the control program for each dispatchable unit. The linkage stack is used to save the execution state and the contents of the general registers and access registers during the BRANCH AND STACK and stacking PROGRAM CALL operations. The linkage stack is also used to restore a portion of the execution state and general-register and access-register contents during the PROGRAM RETURN operation.

A linkage stack resides in virtual storage. The linkage stack for a dispatchable unit is in the home address space for that dispatchable unit. The home address space is designated by the home address-space-control element in control register 13.

The linkage stack is intended to be protected from problem-state programs so that these programs cannot examine or modify the information saved in the linkage stack, except by means of the EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE instructions. This protection can be obtained by means of key-controlled protection.

A linkage stack may consist of a number of linkage-stack sections chained together. A linkage-stack section is variable in length. The maximum length of each linkage-stack section is 65,560 bytes.

There are three types of entry in the linkage stack: header entry, trailer entry, and state entry. A header entry and a trailer entry are at the beginning and end, respectively, of a linkage-stack section, and they are used to chain linkage-stack sections together. Header entries and trailer entries are formed by the control program. A state entry is used to contain the execution state and register contents that are saved during the BRANCH AND STACK or stacking PROGRAM CALL operation, and it is formed during the operation. A state entry is further distinguished as
being a branch state entry if it was formed by BRANCH AND STACK or as being a program-call state entry if it was formed by PROGRAM CALL.

The actions of forming a state entry and saving information in it during the BRANCH AND STACK and stacking PROGRAM CALL operations are called the stacking process. The actions of restoring information from a state entry and logically deleting the entry during the PROGRAM RETURN operation are called the unstacking process. The part of the unstacking process that locates a state entry is also performed during the EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE operations.

Each type of linkage-stack entry has a length that is a multiple of eight bytes. A header entry and trailer entry each has a length of 16 bytes. A state entry has a length of 296 bytes.

Each of the header entry, trailer entry, and state entry has a common eight-byte area at its end, called the entry descriptor. The linkage-stack-entry address in control register 15 designates the leftmost byte of the entry descriptor of the last linkage-stack entry, other than the trailer entry, in a linkage-stack section. This entry is called the current linkage-stack entry, and the section is called the current linkage-stack section.

Each entry descriptor in a linkage-stack section, except the one in the trailer entry of the section, includes a field that specifies the amount of space existing between the end of the entry descriptor and the beginning of the trailer entry. This field is named the remaining-free-space field. The remaining-free-space field in a trailer entry is unused.

When a new state entry is to be formed in the linkage stack during the stacking process, the new entry is placed immediately after the entry descriptor of the current linkage-stack entry, provided that there is enough remaining free space in the current linkage-stack section to contain the new entry. If there is not enough remaining free space in the current section, and if the trailer entry in the current section indicates that another section follows the current section, the new entry is placed immediately after the entry descriptor of the header entry of that following section, provided that there is enough remaining free space in that section. If the trailer entry indicates that there is not a following section, an exception is recognized, and a program interruption occurs. It is then the responsibility of the control program to allocate another section, chain it to the current section, and cause the BRANCH AND STACK or stacking PROGRAM CALL instruction to be reexecuted. If there is a following section but there is not enough remaining free space in it, an exception is recognized.

If the remaining-free-space value that is used to locate a trailer entry is not a multiple of 8, an exception is recognized. The remaining-free-space value in the header entry of a linkage-stack section must be set to a multiple of 8 to ensure that the remaining-free-space value that may be used to locate the trailer entry of the section will be a multiple of 8.

When the stacking process is successful in forming a new state entry, it updates the linkage-stack-entry address in control register 15 so that the address designates the leftmost byte of the entry descriptor of the new entry, which thus becomes the new current linkage-stack entry.

When, during the unstacking process in PROGRAM RETURN, the current linkage-stack entry is a state entry, the process operates on that entry and then updates the linkage-stack-entry address so that it designates the entry descriptor of the preceding entry in the same linkage-stack section. The preceding entry thus becomes the current entry. The new current entry may be another state entry, or it may be a header entry.

The header entry of a linkage-stack section indicates whether there is a preceding section. If there is a preceding section, the header entry contains the address of the last linkage-stack entry, other than the trailer entry, in the preceding section. That last entry should be a state entry (not another header entry), unless there is an error in the linkage stack.

If the unstacking process is performed when the current linkage-stack entry is a header entry, and if the header entry indicates that a preceding linkage-stack section exists, the unstacking process proceeds by treating the entry designated in the preceding section as if it were the current entry, provided that this entry is a state entry. If the header entry does not indicate a preceding
section, or if the entry designated in the preceding section is not a state entry, an exception is recognized.

When the unstacking process is performed in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the process locates a state entry but does not change the linkage-stack-entry address in control register 15.

Each entry descriptor in a linkage-stack section includes a field that specifies the length of the next linkage-stack entry, other than the trailer entry, in the section. When a state entry is created during the stacking process, zeros are placed in this field in the created entry, and the length of the state entry is placed in this field in the preceding entry. When a state entry is logically deleted during the unstacking process in PROGRAM RETURN, zeros are placed in this field in the preceding entry. This field is named the next-entry-size field.

When the stacking or unstacking process operates on the linkage stack, key-controlled protection does not apply, but low-address and page protection do apply.

Linkage-Stack-Operations Control

The use of the linkage stack is controlled by the ASN-and-LX-reuse control in control register 0, the home address-space-control element in control register 13 and the linkage-stack-entry address in control register 15. The home address-space-control element is described in “Dynamic Address Translation” on page 3-33. The ASN-and-LX-reuse control and the linkage-stack-entry address are described below.

Control Register 0

| R |
| 44 |

ASN-and-LX-Reuse Control (R): Bit 44 of control register 0 is the ASN-and-LX-reuse-control bit (R) and is assigned if the ASN-and-LX-reuse facility is installed. The effects of a one value of the bit specifically on the linkage stack are as follows:

- BRANCH AND STACK and stacking PROGRAM CALL save the secondary ASN-second-table-entry instance number (ASTEIN), bits 0-31 of control register 3, and the primary ASTEIN, bits 0-31 of control register 4, in bytes 176-179 and 180-183, respectively, of the linkage-stack state entry that either of the instructions forms.

- PROGRAM RETURN with space switching, after it has used the PASN in bytes 134 and 135 of the state entry to locate an ASN-second-table entry, compares the primary ASTEIN in bytes 180-183 of the state entry to the ASTEIN in the ASN-second-table entry. PROGRAM RETURN to current primary or with space switching, if it uses the SASN in bytes 130 and 131 of the state entry to locate an ASN-second-table entry (because the new SASN is not equal to the new PASN), compares the secondary ASTEIN in bytes 176-179 of the state entry to the ASTEIN in the ASN-second-table entry. The one or two comparisons of ASTEINs must each give equal results; otherwise, an ASTE-instance exception is recognized. The comparisons occur regardless of the reusable-ASN bits in the ASN-second-table entries. PROGRAM RETURN to current primary or with space switching restores the secondary and primary ASTEINs saved in the state entry to bit positions 0-31 of control registers 3 and 4, respectively (if no exception is recognized). When unstacking a state entry formed by BRANCH AND STACK, PROGRAM RETURN does not compare or restore ASTEINs.

The effect of the ASN-and-LX-reuse control on PC-number translation is described in “ASN-and-LX-Reuse Control (R)” on page 5-32. Other effects of the control are described in “ASN-Second-Table-Entry Instance Number and ASN Reuse” on page 3-19.

Control Register 15

The location of the entry descriptor of the current linkage-stack entry is specified in control register 15. The register has the following format:

<table>
<thead>
<tr>
<th>Linkage-Stack-Entry Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 61 63</td>
</tr>
</tbody>
</table>
**Linkage-Stack-Entry Address:** Bits 0-60 of control register 15, with three zeros appended on the right, form the 64-bit home virtual address of the entry descriptor of the current linkage-stack entry in the current linkage-stack section. Bits 0-60 are changed during the stacking process in BRANCH AND STACK and stacking PROGRAM CALL and during the unstacking process in PROGRAM RETURN. Bits 61-63 of control register 15 are set to zeros when bits 0-60 are changed.

**Linkage Stack**

The linkage stack consists of one or more linkage-stack sections containing linkage-stack entries. There are three principal types of linkage-stack entry: header entry, trailer entry, and state entry. A state entry is further distinguished as being either a branch state entry or a program-call state entry.

Each type of linkage-stack entry has an entry descriptor at its end. The leftmost byte of the entry descriptor of the current linkage-stack entry in the current linkage-stack section is designated by the linkage-stack-entry address in control register 15.

The linkage stack resides in the home address space, designated by the home address-space-control element in control register 13.

**Entry Descriptors**

An entry descriptor is at the end of each linkage-stack entry. The entry descriptor is eight bytes in length and has the following format:

```
 U | ET | SI | RFS | NES |
```

0 1 8 16 32 48 63

The fields in the entry descriptor are allocated as follows:

**Unstack-Suppression Bit (U):** When bit 0 is one in the entry descriptor of a header entry or state entry encountered during the unstacking process in PROGRAM RETURN, a stack-operation exception is recognized. Bit 0 is ignored in a trailer entry and during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE. The control program can temporarily set bit 0 to one in the current linkage-stack entry (a header entry or state entry) to prevent PROGRAM RETURN from being executed successfully while still allowing EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE to be executed successfully. Bit 0 is set to zero in the entry descriptor of a state entry when the entry is formed during the stacking process.

**Entry Type (ET):** Bits 1-7 are a code that specifies the type of the linkage-stack entry containing the entry descriptor. The assigned codes are:

<table>
<thead>
<tr>
<th>Code (in Binary)</th>
<th>Entry Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001001</td>
<td>Header entry</td>
</tr>
<tr>
<td>0001010</td>
<td>Trailer entry</td>
</tr>
<tr>
<td>0001100</td>
<td>Branch state entry</td>
</tr>
<tr>
<td>0001101</td>
<td>Program-call state entry</td>
</tr>
</tbody>
</table>

Codes 0000000-0001000, 0001011, and 0001110 through 0111111 binary are reserved for possible future assignments. Codes 1000000 through 1111111 binary are available for use by programming.

Bits 1-7 are set to 0001100 or 0001101 binary in the entry descriptor of a state entry when the entry is formed during the stacking process.

A stack-type exception is recognized during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN if bits 1-7 in the current linkage-stack entry do not indicate that the entry is a state entry or a header entry; or, when the current entry is a header entry, if bits 1-7 in the entry designated by the backward stack-entry address in the header entry do not indicate that the designated entry is a state entry. However, a stack-specification exception is recognized, instead of a stack-type exception, if both the current entry and the designated entry are header entries.

**Section Identification (SI):** Bits 8-15 are an identification, provided by the control program, of the linkage-stack section containing the entry descriptor. In the state entry formed by a stacking process, the process sets bits 8-15 equal to the contents of the section-identification field in the preceding linkage-stack entry.
**Remaining Free Space (RFS):** Bits 16-31 specify the number of bytes between the end of this entry descriptor and the beginning of the trailer entry in the same linkage-stack section, except that this field in a trailer entry has no meaning. Thus, in the last state entry in a section, or in the header entry if there is no state entry, bits 16-31 specify the number of bytes available in the section for performance of the stacking process. In the state entry formed by a stacking process, the process sets bits 16-31 equal to the contents of the remaining-free-space field in the preceding linkage-stack entry minus the size, in bytes, of the new entry. Bits 16-31 must be a multiple of 8 (bits 29-31 must be zeros) in the entry descriptor of the header entry in a linkage-stack section; otherwise, a value that is not a multiple of 8 will be propagated to bits 16-31 in the entry descriptor of each state entry in the section, and a stack-specification exception will be recognized if the stacking process attempts to locate the trailer entry in the section in order to proceed to the next section.

**Next-Entry Size (NES):** Bits 32-47 specify the size in bytes of the next linkage-stack entry, other than a trailer entry, in the same linkage-stack section. This field in the current linkage-stack entry contains all zeros. This field in a trailer entry has no meaning. When the stacking process forms a state entry, it places zeros in the next-entry-size field of the new entry, and it places the size of the new entry in the next-entry-size field of the preceding entry. When the unstacking process logically deletes a state entry, it places zeros in the next-entry-size field of the preceding entry, which entry becomes the current entry.

Bits 48-63 are set to zeros in a state entry when the entry is formed during the stacking process. In a header entry, trailer entry, or state entry, bits 48-63 are reserved for possible future extensions and should always be zeros.

**Programming Note:** No entry-type code will be assigned in which the leftmost bit of the code is one. The control program can temporarily set the leftmost bit to one in the entry-type code of the current linkage-stack entry (a header entry or a state entry) to prevent the successful execution of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN.

**Header Entries**
A header entry is at the beginning of each linkage-stack section. The header entry is 16 bytes in length and has the following format:

```
+------------+----+
| BSEA       | B  |
+------------+----+
| 0          | 61 |
```

```
Entry Descriptor
```

```
64 127
```

The fields in the first eight bytes of the header entry are allocated as follows:

**Backward Stack-Entry Validity Bit (B):** Bit 63, when one, specifies that the preceding linkage-stack section is available and that the backward stack-entry address, bits 0-60, is valid. Bit 63 is set to one during the stacking process when the process proceeds to this section from the preceding one because there is not enough space available in the preceding section to perform the process. During the unstacking process when this header entry is the current linkage-stack entry, a stack-empty exception is recognized if bit 63 is zero.

**Backward Stack-Entry Address (BSEA):** When bit 63 is one, bits 0-60 with three zeros appended on the right, form the 64-bit home virtual address of the entry descriptor of the last linkage-stack entry, other than the trailer entry, in the preceding linkage-stack section. However, if the current linkage-stack entry is in the preceding or an earlier linkage-stack section, bits 0-60 may have no meaning because the entry they designate, and earlier entries, may have been logically deleted. Bits 0-60 are set during the stacking process when the process proceeds to this section from the preceding one because there is not enough space available in the preceding section to perform the process. During the unstacking process when this header entry is the current linkage-stack entry and bit 63 is one, the entry is treated as the current entry. 0-60 is treated as the current entry.

Bits 61 and 62 are set to zeros when bits 0-60 are set during the stacking process. Bits 61 and 62 are reserved for possible future extensions.
**Trailer Entries**

A trailer entry is at the end of each linkage-stack section. The trailer entry begins immediately after the area specified by the remaining-free-space field in the entry descriptors of the header entry and each state entry in the same linkage-stack section. The trailer entry is 16 bytes in length and has the following format:

```
 ┌──────────────────────────────┬──┬─┐
 │ FSHA │ │F│
 └──────────────────────────────┴──┴─┘
```

The fields in the first eight bytes of the trailer entry are allocated as follows:

**Forward-Section Validity Bit (F):** Bit 63, when one, specifies that the next linkage-stack section is available and that the forward-section-header address, bits 0-60, is valid. During the stacking process when there is not enough space available in the current linkage-stack section to perform the process, a stack-full exception is recognized if bit 63 in the trailer entry of the current section is zero.

**Forward-Section-Header Address (FSHA):** When bit 63 is one, bits 0-60, with three zeros appended on the right, form the 64-bit home virtual address of the entry descriptor of the header entry in the next linkage-stack section. During the stacking process when there is not enough space available in the current section to perform the process and bit 63 is one, the header entry designated by bits 0-60 becomes the current linkage-stack entry.

Bits 61 and 62 are reserved for possible future extensions.

**Programming Note:** All of the fields in the trailer entry are set only by the control program.

---

**State Entries**

Zero, one, or more state entries may follow the header entry in each linkage-stack section. A state entry may be a branch state entry, formed by a BRANCH AND STACK instruction, or a program-call state entry, formed by a stacking PROGRAM CALL instruction. The state entry is 296 bytes in length and has the following format:

```
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>70</td>
<td>112</td>
</tr>
<tr>
<td>78</td>
<td>120</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>128</td>
</tr>
<tr>
<td>88</td>
<td>136</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>288</td>
</tr>
</tbody>
</table>
```

Bytes 0-127 of the state entry contain the contents of general registers 0-15 in the ascending order of the register numbers. Bytes 224-287 contain the contents of access registers 0-15 in the ascending order of the register numbers. The contents of these fields are moved from the registers to the state entry during the BRANCH AND STACK and stacking PROGRAM CALL operations. The contents of general registers 2-14 and access registers 2-14 are restored from the state entry to the registers during the PROGRAM RETURN operation. The contents of a specified range of general registers and access registers can be restored from the state entry to the registers by EXTRACT STACKED REGISTERS.

Bytes 128-223 of the state entry contain the other status information that is placed in the entry by BRANCH AND STACK, stacking PROGRAM CALL, and MODIFY STACKED STATE. A portion of this status information is restored to the PSW and control registers by PROGRAM RETURN, and all of the information can be examined by means of EXTRACT STACKED STATE. Bytes 288-295 contain the entry descriptor. EXTRACT STACKED STATE sets the condition code to indi-
cate whether the entry-type code in the entry descriptor specifies a branch state entry or a program-call state entry.

Bytes 128-223 of the state entry have the following detailed format:

<table>
<thead>
<tr>
<th>PKM</th>
<th>SASN</th>
<th>EAX</th>
<th>PASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>130</td>
<td>132</td>
<td>134</td>
</tr>
<tr>
<td>135</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSW Bits 0-63

136 143

In a Branch State Entry Made in 24-Bit or 31-Bit Mode

<table>
<thead>
<tr>
<th>Bits 33-63 of Branch Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>144 148 151</td>
</tr>
</tbody>
</table>

In a Branch State Entry Made in 64-Bit Mode

<table>
<thead>
<tr>
<th>Bits 0-62 of Branch Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>144 151</td>
</tr>
</tbody>
</table>

In a Program-Call State Entry Made when Resulting Mode Is 24 Bit or 31 Bit

<table>
<thead>
<tr>
<th>Called-Space Id. 0</th>
<th>Numeric Part of PC Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>144 148 151</td>
<td></td>
</tr>
</tbody>
</table>

In a Program-Call State Entry Made when Resulting Mode Is 64 Bit

<table>
<thead>
<tr>
<th>Called-Space Id. 1</th>
<th>Numeric Part of PC Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>144 148 151</td>
<td></td>
</tr>
</tbody>
</table>

Modifiable Area

152 159

All Zeros

160 167

PSW Bits 64-127

168 175

If ASN-and-LX Reuse Is Enabled; otherwise Unpredictable

<table>
<thead>
<tr>
<th>Secondary ASTEIN</th>
<th>Primary ASTEIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>176 180 183</td>
<td></td>
</tr>
</tbody>
</table>

The fields in bytes 128-183 are allocated as follows. In the following, “of the calling program" means the value existing at the beginning of the execution of the BRANCH AND STACK or stacking PROGRAM CALL instruction that formed the state entry.

**PSW-Key Mask (PKM):** Bytes 128-129 contain the PSW-key mask, bits 32-47 of control register 3, of the calling program. The PSW-key mask is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.

**Secondary ASN (SASN):** Bytes 130-131 contain the secondary ASN, bits 48-63 of control register 3, of the calling program. The SASN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.

**Extended Authorization Index (EAX):** Bytes 132-133 contain the extended authorization index, bits 32-47 of control register 8, of the calling program. The EAX is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.

**Primary ASN (PASN):** Bytes 134-135 contain the primary ASN, bits 48-63 of control register 4, of the calling program. The PASN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL, and it is restored to the control register by a PROGRAM RETURN instruction that unstacks an entry formed by stacking PROGRAM CALL.
**Program-Status Word (PSW):** In a branch state entry made by a BRANCH AND STACK instruction in which the R₁ field is zero, and in a program-call state entry, bytes 136-143 and 168-175 contain the updated PSW of the calling program. Bytes 136-143 contain bits 0-63 of the PSW, and bytes 168-175 contain bits 64-127 of the PSW. Thus, the basic and extended addressing-mode bits in this PSW specify the addressing mode of the calling program, and the instruction address designates the next sequential instruction following the BRANCH AND STACK or stacking PROGRAM CALL instruction that formed the state entry, or following an EXECUTE instruction that had the BRANCH AND STACK or stacking PROGRAM CALL instruction as its target instruction. In a branch state entry formed by a BRANCH AND STACK instruction in which the R₁ field is nonzero, bytes 136-143 and 168-175 contain the PSW of the calling program, except that the extended-addressing-mode bit in bit position 31 of bytes 136-139, the basic-addressing-mode bit in bit position 0 of byte 140, and the instruction address in bytes 168-175 are as specified by the contents of the general register designated by the R₁ field. See the definition of BRANCH AND STACK in Chapter 10, “Control Instructions” for how the basic- and extended-addressing-mode bits and instruction address are specified. The value of the PER mask in bytes 136-143 is always unpredictable. The PSW is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL and is restored as the current PSW by PROGRAM RETURN, except that the PER mask is not restored. PROGRAM RETURN does not change the PER mask in the current PSW.

**Basic Addressing Mode (A):** In a branch state entry made in the 24-bit or 31-bit addressing mode, bit position 0 of bytes 148-151 contains the basic-addressing-mode bit, bit 32 of the PSW, at the end of the execution of the BRANCH AND STACK instruction that formed the state entry. The basic-addressing-mode bit is saved in bit position 0 of bytes 148-151 by BRANCH AND STACK. BRANCH AND STACK does not change the basic-addressing-mode bit in the PSW.

**Branch Address:** In a branch state entry made in the 24-bit or 31-bit addressing mode, bit positions 1-31 of bytes 148-151 contain bits 33-63 of the instruction address in the PSW at the end of the execution of the BRANCH AND STACK instruction that formed the state entry, and the contents of bytes 144-147 are unpredictable. In a branch state entry made in the 64-bit addressing mode, bytes 144-151 contain bits 0-62 of that instruction address with a one appended on the right. The instruction address is saved in bytes 148-151 or 144-151 (depending on the addressing mode) by BRANCH AND STACK. When the R₂ field of BRANCH AND STACK is nonzero, the instruction causes branching, and the instruction address in bytes 148-151 or 144-151 is the branch address. When the R₂ field of BRANCH AND STACK is zero, the instruction is executed without branching, and the instruction address in bytes 148-151 or 144-151 is the address of the next sequential instruction following the BRANCH AND STACK instruction, or following an EXECUTE instruction that had the BRANCH AND STACK instruction as its target instruction.

**Called-Space Identification:** In a program-call state entry, bytes 144-147 contain the called-space identification (CSI). The CSI is saved in the state entry by stacking PROGRAM CALL.

If the PROGRAM CALL operation was space switching, bytes 0 and 1 of the CSI (bytes 144 and 145 of the state entry) contain the new primary ASN that was placed in control register 4 by the PROGRAM CALL instruction. When the ASN-and-LX-reuse facility is not enabled, bytes 2 and 3 of the CSI (bytes 146 and 147 of the state entry) contain the rightmost two bytes of the ASTE sequence number (ASTESN) in the new primary ASTE whose address was placed in control register 5 by the PROGRAM CALL instruction. When the ASN-and-LX-reuse facility is installed and enabled, bytes 2 and 3 of the CSI (bytes 146 and 147 of the state entry) contain the rightmost two bytes of the ASTE instance number (ASTEIN) in the new primary ASTE whose address was placed in control register 5 by the PROGRAM CALL instruction.

If the PROGRAM CALL operation was the to-current-primary operation, the CSI is all zeros.

**Numeric Part of PC Number:** In a program-call state entry, bit positions 1-31 of bytes 148-151 contain the numeric part of the PC number used by the stacking PROGRAM CALL instruction that formed the entry. When ASN-and-LX reuse is not enabled, or when it is and bit 44 of the effective address used by stacking PROGRAM CALL is
zero, stacking PROGRAM CALL places bits 44-63 of the effective address, with 11 zeros appended on the left, in bit positions 1-31 of bytes 148-151. When ASN-and-LX reuse is enabled and bit 44 of the effective address is one, stacking PROGRAM CALL places bits 45-63 of the effective address, with bits 32-43 of the effective address appended on the left, in bit positions 1-31 of bytes 148-151. In any case, stacking PROGRAM CALL places a zero in bit position 0 of the bytes if the resulting addressing mode is the 24-bit or 31-bit mode or a one in bit position 0 if the resulting addressing mode is the 64-bit mode.

Modifiable Area: Bytes 152-159 are the field that is set by MODIFY STACKED STATE. BRANCH AND STACK and stacking PROGRAM CALL place all zeros in bytes 152-159.

Secondary ASTEIN (SASTEIN): If the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control in control register 0, bytes 176-179 contain the secondary ASTEIN, bits 0-31 of control register 3, of the calling program. The SASTEIN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL. When PROGRAM RETURN unstacks a program-call state entry, it compares the SASTEIN in the state entry to the ASTEIN in the ASN-second-table entry if SASN translation occurs, and it always restores the SASTEIN to control register 3.

Primary ASTEIN (PASTEIN): If the ASN-and-LX-reuse facility is installed and is enabled by the ASN-and-LX-reuse control in control register 0, bytes 180-183 contain the primary ASTEIN, bits 0-31 of control register 4, of the calling program. The PASTEIN is saved in the state entry by BRANCH AND STACK or stacking PROGRAM CALL. When PROGRAM RETURN unstacks a program-call state entry, it compares the PASTEIN in the state entry to the ASTEIN in the ASN-second-table entry if PASN translation occurs, and it always restores the PASTEIN to control register 4.

All zeros are placed in bytes 160-167 by BRANCH AND STACK and stacking PROGRAM CALL.

The contents of bytes 184-223 are unpredictable.

Stacking Process

The stacking process is performed as part of a BRANCH AND STACK or stacking PROGRAM CALL operation. The process locates space for a new linkage-stack state entry, forms the entry, updates the next-entry-size field in the preceding entry, and updates the linkage-stack-entry address in control register 15 so that the new entry becomes the current linkage-stack entry.

For the stacking process to be performed successfully, DAT must be on and the CPU must be in the primary-space mode or access-register mode; otherwise, a special-operation exception is recognized, and the operation is suppressed.

Except as just mentioned, the stacking process is performed independent of the current addressing mode and translation mode, as specified by bits 31, 32, 16, and 17 of the current PSW. All addresses used during the stacking process are always 64-bit home virtual addresses.

During the stacking process when any address is formed through the addition or subtraction of a value to or from another address, a carry out of, or a borrow into, bit position 0 of the address, if any, is ignored.

When the stacking process fetches or stores by using an address that designates, after translation, a location that is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Key-controlled protection does not apply to the accesses made during the stacking process, but page protection and low-address protection do apply. A protection exception causes the operation to be suppressed.

Locating Space for a New Entry

The linkage-stack-entry address in control register 15 is used to locate the current linkage-stack entry. Bits 0-60 of control register 15, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the current linkage-stack entry.

The first word of the entry descriptor of the current linkage-stack entry is fetched by using the 64-bit home virtual address. This fetch is for the purpose of obtaining the section-identification and
remaining-free-space fields in the word; the unstack-suppression bit and entry-type field in the word are not examined.

The 16-bit unsigned binary value in the remaining-free-space field, bits 16-31 of the entry descriptor, is compared against the size in bytes of the linkage-stack entry to be formed. The size of a state entry is 296 bytes. If the value in the field is equal to or greater than the size of the entry to be formed, processing continues as described in "Forming the New Entry" otherwise, processing continues as described below.

When the remaining-free-space field in the current linkage-stack entry indicates that there is not enough space available in the current linkage-stack section to form the new entry, the first doubleword of the trailer entry of the current section is fetched. The address for fetching this doubleword is determined as follows: to the address formed from the contents of control register 15, add 8 to address the first byte after the entry descriptor of the current entry, and then add the contents of the remaining-free-space field of the current entry to address the first byte of the trailer entry. The remaining-free-space value used in the addition must be a multiple of 8; otherwise, a stack-specification exception is recognized, and the operation is nullified.

If the forward-section-validity bit, bit 63, of the trailer entry is zero, a stack-full exception is recognized, and the operation is nullified; otherwise, the forward-section-header address in the trailer entry is used to locate the header entry in the next linkage-stack section. Bits 0-60, of the trailer entry, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the header entry in the next section.

The first word of the entry descriptor of the header entry in the next linkage-stack section is fetched. This fetch is for the purpose of obtaining the section-identification and remaining-free-space fields in the word; the unstack-suppression bit and entry-type field in the word are not examined.

The value in the remaining-free-space field of the header entry in the next linkage-stack section is compared against the size in bytes of the entry to be formed. If the value in the field is equal to or greater than the size of the entry to be formed, the following occurs:

- The linkage-stack-entry address, bits 0-60 of control register 15, is placed, as the backward stack-entry address, in bit positions 0-60 of the header entry in the next linkage-stack section, and zeros are placed in bit positions 61 and 62.
- The backward stack-entry validity bit, bit 63, in the header entry in the next section is set to one.
- Bits 0-60 of the 64-bit home virtual address of the entry descriptor of the header entry in the next section are placed in bit positions 0-60 of control register 15, and zeros are placed in bit positions 61-63. of control register 15. Thus, the header entry in the next section becomes the current linkage-stack entry, and the next section becomes the current linkage-stack section.
- Processing continues as described in "Forming the New Entry."

If the value in the remaining-free-space field of the header entry in the next section (before the next section becomes the current section) is less than the size of the linkage-stack entry to be formed, a stack-specification exception is recognized, and the operation is nullified.

**Forming the New Entry**

When the remaining-free-space field in the current linkage-stack entry indicates that there is enough space available in the current linkage-stack section to form the new entry, the new entry is formed beginning immediately after the entry descriptor of the current entry.

The new entry is a state entry. The contents of general registers 0-15 are stored in bytes 0-127 of the new entry, in the ascending order of the register numbers. The contents of access registers 0-15 are stored in bytes 224-287 of the new entry, in the ascending order of the register numbers. The PSW-key mask, bits 32-47 of control register 3; secondary ASN, bits 48-63 of control register 3; extended authorization index, bits 32-47 of control register 8; and primary ASN, bits 48-63 of control register 4, are stored in bytes 128-129, 130-131, 132-133, and 134-135, respectively, of the new entry. The current PSW, in which the instruction address has been updated, is stored in bytes
136-143 and 168-175 of the new entry. Bytes 136-143 contain bits 0-63 of the PSW, and bytes 168-175 contain bits 64-127 of the PSW. However, the value of the PER mask, bit 1 in the PSW stored, is unpredictable. Also, if the instruction being executed is a BRANCH AND STACK instruction in which the R field is nonzero, the extended- and basic-addressing-mode bits stored in bytes 136 and 140, respectively, of the new entry, and the instruction address stored in bytes 168-175 of the new entry, are as specified by the contents of the general register designated by the R field.

If the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the secondary ASTEIN, bits 0-31 of control register 3, and the primary ASTEIN, bits 0-31 of control register 4, are stored in bytes 176-179 and 180-183, respectively, of the new entry.

When the instruction is PROGRAM CALL, the called-space identification is stored in bytes 144-147 of the new entry. When the instruction is performing the space-switching PROGRAM CALL operation and the ASN-and-LX-reuse facility is not enabled, the called-space identification is the two-byte ASN, bytes 10 and 11, in the entry-table entry used by the instruction, followed by bytes 2 and 3 of the ASTE sequence number, bytes 2 and 3 being bits 176-191, in the ASN-second-table entry specified by the ASN. When the instruction is performing the space-switching PROGRAM CALL operation and the ASN-and-LX-reuse facility is installed and enabled, the called-space identification is the two-byte ASN, bytes 10 and 11, in the entry-table entry used by the instruction, followed by bytes 2 and 3 of the ASTE instance number, bytes 2 and 3 being bits 368-383, in the ASN-second-table entry specified by the ASN. When the instruction is performing the to-current-primary PROGRAM CALL operation, the called-space identification is all zeros.

When the instruction is BRANCH AND STACK in the 24-bit or 31-bit addressing mode, the basic-addressing-mode bit from the current PSW is stored in bit position 0 of byte 148 in the state entry, bits 33-63 of the branch address, or of the updated instruction address if the operation is performed without branching, are stored in bit positions 1-31 of bytes 148-151, and the contents of bytes 144-147 are unpredictable. In the 64-bit addressing mode, bits 0-62 of the branch address or updated instruction address, with a one appended on the right, are stored in bytes 144-151 of the state entry.

When the instruction is PROGRAM CALL, the numeric part of the PC number used, with 11 zeros appended on the left if the number is 20 bits, is stored in bit positions 1-31 of bytes 148-151. If the resulting addressing mode after the execution of PROGRAM CALL is the 24-bit or 31-bit addressing mode, a zero is stored in bit position 0 of byte 148. If the resulting addressing mode is the 64-bit addressing mode, a one instead of a zero is stored in bit position 0 of byte 148.

Zeros are stored in bytes 152-167 of the new entry. The contents of bytes 176-223 are unpredictable, or the contents of bytes 184-223 are unpredictable if the secondary ASTEIN and primary ASTEIN were stored.

Bytes 288-295 of the new entry are its entry descriptor. The unstack-suppression bit, bit 0, of this entry descriptor is set to zero. The code 0001100 binary is stored in the entry-type field, bits 1-7, of this entry descriptor if the instruction being executed is BRANCH AND STACK. The code 0001101 binary is stored if the instruction is PROGRAM CALL. The value in the section-identification field of the current linkage-stack entry is stored in the section-identification field, bits 8-15, of this entry descriptor. The value in the remaining-free-space field of the current entry, minus the size in bytes of the new entry, is stored in the remaining-free-space field of this entry descriptor. Zeros are stored in the next-entry-size field, bits 32-47, and in bit positions 48-63 of this entry descriptor.

The stores into the new entry appear to be word concurrent as observed by other CPUs. The order in which the stores occur is unpredictable.

**Updating the Current Entry**

The size in bytes of the new linkage-stack entry is stored in the next-entry-size field of the current entry. The remainder of the current entry remains unchanged.

The order of the stores into the current entry and the new entry is unpredictable.
Updating Control Register 15
Bits 0-60 of the 64-bit home virtual address of the entry descriptor of the new linkage-stack entry are placed in bit positions 0-60 of control register 15, the linkage-stack-entry address. Zeros are placed in bit positions 61-63 of control register 15. Thus, the new entry becomes the current linkage-stack-entry.

Recognition of Exceptions during the Stacking Process
The exceptions which can be encountered during the stacking process and their priority are described in the definitions of the BRANCH AND STACK and PROGRAM CALL instructions.

Programming Note: Any exception recognized during the execution of BRANCH AND STACK and PROGRAM CALL causes either nullification or suppression. Therefore, if an exception is recognized, the stacking process does not store into any linkage-stack entry or change the contents of control register 15.

Unstacking Process
The unstacking process is performed as part of the PROGRAM RETURN operation. The process locates the last state entry in the linkage stack, restores a portion of the information in the entry to the CPU registers, updates the next-entry-size field in the preceding entry, and updates the linkage-stack-entry address in control register 15 so that the preceding entry becomes the current linkage-stack entry. The part of the unstacking process that locates the last state entry is also performed as part of the EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE operations.

For the unstacking process to be performed successfully, DAT must be on and the CPU must be in the primary-space mode or access-register mode; otherwise, a special-operation exception is recognized, and the operation is suppressed. However, when the unstacking process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the CPU may be in the primary-space, access-register, or home-space mode.

Except as just mentioned, the unstacking process is performed independent of the current addressing mode and translation mode, as specified by bits 31, 32, 16, and 17 of the current PSW. All addresses used during the unstacking process are always 64-bit home virtual addresses.

During the unstacking process when any address is formed through the addition or subtraction of a value to or from another address, a carry out of, or a borrow into, bit position 0 of the address, if any, is ignored.

When the unstacking process fetches or stores by using an address that designates, after translation, a location that is not available in the configuration, an addressing exception is recognized, and the operation is suppressed.

Key-controlled protection does not apply to the accesses made during the unstacking process, but page protection and low-address protection do apply. A protection exception causes the operation to be suppressed.

Locating the Current Entry and Processing a Header Entry
The linkage-stack-entry address in control register 15 is used to locate the current linkage-stack entry. Bits 0-60 of control register 15, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the current linkage-stack entry.

The first word of the entry descriptor of the current linkage-stack entry is fetched by using the 64-bit home virtual address. If the entry-type code in bits 1-7 of the entry descriptor is not 0001001 binary, indicating that the entry is not a header entry, processing continues as described in “Checking for a State Entry” on page 5-87; otherwise, processing continues as described below.

When the entry-type code in the current linkage-stack entry is 0001001 binary, indicating a header entry, the next processing depends on which instruction is being executed. When the unstacking process is performed as part of the PROGRAM RETURN operation and the unstack-suppression bit, bit 0, in the entry descriptor of the current entry is one, a stack-operation exception is recognized, and the operation is nullified. When the unstacking process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the unstack-suppression bit is ignored.
When there is not an exception due to the unstack-suppression bit, the first doubleword of the current linkage-stack entry (a header entry) is fetched. The address of this doubleword is determined by subtracting 8 from the address of the entry descriptor of the current entry.

If the backward stack-entry validity bit, bit 63, of the current entry is zero, a stack-empty exception is recognized, and the operation is nullified; otherwise, the backward stack-entry address in the current entry is used to locate a linkage-stack entry referred to here as the designated entry. Bits 0-60 of the current entry, with three zeros appended on the right, form the 64-bit home virtual address of the leftmost byte of the entry descriptor of the designated entry.

It is assumed in this definition of the unstacking process that the designated linkage-stack entry is the last entry, other than the trailer entry, in the preceding linkage-stack section. This assumption does not imply any processing that is not explicitly described.

The first word of the entry descriptor of the designated entry is fetched. If the entry-type code in this entry descriptor is not 0001001 binary, indicating that the entry is not a header entry, the following occurs:

- When the unstacking process is performed as part of the PROGRAM RETURN operation, bits 0-60 of the 64-bit home virtual address of the entry descriptor of the designated entry are placed in bit positions 0-60 of control register 15, and zeros are placed in bit positions 61-63 of control register 15. Thus, the designated entry becomes the current linkage-stack entry, and the preceding section (based on the assumption) becomes the current linkage-stack section. When the unstacking process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the process is completed; that is, no additional processing occurs as a part of the unstacking process.

- If the current linkage-stack entry is a state entry, the next processing depends on which instruction is being executed. When the unstacking process is performed as part of the PROGRAM RETURN operation, processing continues as described in "Restoring Information." When the process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the process is completed; that is, no additional processing occurs as a part of the unstacking process.

- If the current linkage-stack entry is not a state entry (and necessarily not a header entry either), a stack-type exception is recognized, and the operation is nullified.

Checking for a State Entry
When the entry-type code in the current linkage-stack entry indicates that the entry is not a header entry, the code is checked for being 0001100 or 0001101 binary, which are the codes assigned to a state entry.

If the current linkage-stack entry is a state entry, the next processing depends on which instruction is being executed. When the unstacking process is performed as part of the PROGRAM RETURN operation, processing continues as described in "Restoring Information." When the process is performed as part of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, or MODIFY STACKED STATE, the process is completed; that is, no additional processing occurs as a part of the unstacking process.

- If the current linkage-stack entry is not a state entry (and necessarily not a header entry either), a stack-type exception is recognized, and the operation is nullified.

Restoring Information
The remaining parts of the unstacking process occur only in the PROGRAM RETURN operation.

The current linkage-stack entry is a state entry. If the unstack-suppression bit in the entry is one, a stack-operation exception is recognized, and the operation is nullified.

When there is not an exception due to the unstack-suppression bit, a portion of the contents of the current linkage-stack entry are restored to the CPU registers. The contents of general registers 2-14 and access registers 2-14 are restored to those registers from where they were saved in the current entry by the stacking process. When the entry-type code in the current entry is 0001101 binary, indicating a program-call state entry, the PSW-key mask and secondary ASN in control register 3, extended authorization index in control register 8, and primary ASN in control register 4 are similarly restored. During this restoration, the authorization index in control register 4 and the monitor masks in control register 8 remain unchanged. (The authorization index may be changed by the part of the PROGRAM RETURN
execution that occurs after the unstacking process.)

When the state entry is a program-call state entry, and if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control in control register 0, the secondary ASTEIN in control register 3 and the primary ASTEIN in control register 4 are restored from where they were saved in the state entry. However, the primary ASTEIN and secondary ASTEIN may be subject to certain tests, as described in the PROGRAM RETURN definition.

When the entry-type code is 0001100 binary, indicating a branch state entry, the PSW-key mask, secondary ASN, extended authorization index, primary ASN, secondary ASTEIN, and primary ASTEIN in the current entry are ignored, and all contents of the control registers remain unchanged.

When the current entry is either a branch state entry or a program-call state entry, bits 0-63 and 64-127 of the current PSW are restored from bytes 136-143 and bytes 168-175, respectively, of the entry, except that the PER mask is not restored. The PER mask in the current PSW remains unchanged. Bytes 144-159 and bytes 160-167 of the current entry are ignored.

The fetches from the current entry appear to be word concurrent as observed by other CPUs. The order in which the fetches occur is unpredictable.

Updating the Preceding Entry
Zeros are stored in the next-entry-size field, bits 32-47, of the entry descriptor of the preceding linkage-stack entry. The remainder of the preceding entry remains unchanged. The address of the entry descriptor of the preceding entry is determined by subtracting the size in bytes of the current entry from the address of the entry descriptor of the current entry.

The order of the store into the preceding entry and the fetches from the current entry is unpredictable.

Updating Control Register 15
Bits 0-60 of the 64-bit home virtual address of the entry descriptor of the preceding linkage-stack entry are placed in bit positions 0-60 of control register 15, the linkage-stack-entry address. Zeros are placed in bit positions 61-63 of control register 15. Thus, the preceding entry becomes the current linkage-stack entry.

Recognition of Exceptions during the Unstacking Process
The exceptions which can be encountered during the unstacking process and their priority are described in the definition of the PROGRAM RETURN instruction. The exceptions which apply to EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, and MODIFY STACKED STATE are described in the definitions of those instructions.

Programming Notes:
1. Any exceptions recognized during the execution of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN cause either nullification or suppression. Therefore, if an exception is recognized, the unstacking process does not change the contents of any CPU register (except for updating the instruction address in the PSW in the case of suppression) or store into any linkage-stack entry.

2. The unstacking process in PROGRAM RETURN does not restore the PER mask in the PSW so that an act of turning PER on or off after the execution of the related BRANCH AND STACK or PROGRAM CALL instruction but before the execution of the PROGRAM RETURN instruction will not be counteracted. When PROGRAM CALL or PROGRAM RETURN is space switching, the space-switch event can be used as a signal to turn PER on or off, if desired.

Sequence of Storage References
The following sections describe the effects which can be observed in storage due to overlapped operations and piecemeal execution of a CPU program. Most of the effects described in these sections are observable only when two or more CPUs or channel programs are in simultaneous
execution and access common storage locations. Thus, most of the effects need be taken into account by a program only if the program interacts with another CPU or a channel program.

Some of the effects described in the following sections are independent of interaction with another CPU or a channel program. These effects, which are therefore more readily observable, relate to prefetched instructions and overlapping operands of a single instruction. These effects are described in "Conceptual Sequence" and in "Interlocks for Virtual-Storage References" on page 5-90.

Conceptual Sequence
In the real mode, primary-space mode, or secondary-space mode, the CPU conceptually processes instructions one at a time, with the execution of one instruction preceding the execution of the following instruction. The execution of the instruction designated by a successful branch follows the execution of the branch. Similarly, an interruption takes place between instructions or, for interruptible instructions, between units of operation of such instructions.

The sequence of events implied by the processing just described is sometimes called the conceptual sequence.

Each operation of instruction execution appears to the program itself to be performed sequentially, with the current instruction being fetched after the preceding operation is completed and before the execution of the current operation is begun. This appearance is maintained even though the storage-implementation characteristics and overlap of instruction execution with storage accessing may cause actual processing to be different. The results generated are those that would have been obtained had the operations been performed in the conceptual sequence. Thus, it is possible for an instruction to modify the next succeeding instruction in storage.

Operations in the access-register mode or home-space mode are the same as in the other translation modes, with one exception: an instruction that is a store-type operand of a preceding instruction may appear to be fetched before the store occurs. Thus, it is not assured that an instruction can modify the succeeding instructions. This exception applies if either the storing instruction or the instruction stored is executed in the access-register or home-space mode.

Regardless of the translation mode, another case in which the copies of prefetched instructions are not necessarily discarded occurs when the fetch and the store are done by means of different effective addresses that map to the same real address. This case is described in more detail in "Interlocks for Virtual-Storage References" on page 5-90.

Overlapped Operation of Instruction Execution
In simple models in which operations are not overlapped, the conceptual and actual sequences are essentially the same. However, in more complex machines, overlapped operation, buffering of operands and results, and execution times which are comparable to the propagation delays between units can cause the actual sequence to differ considerably from the conceptual sequence. In these machines, special circuitry is employed to detect dependencies between operations and ensure that the results obtained, as observed by the CPU which generates them, are those that would have been obtained if the operations had been performed in the conceptual sequence. However, other CPUs and channel programs may, unless otherwise constrained, observe a sequence that differs from the conceptual sequence.

Divisible Instruction Execution
It can normally be assumed that the execution of each instruction occurs as an indivisible event. However, in actual operation, the execution of an instruction consists in a series of discrete steps. Depending on the instruction, operands may be fetched and stored in a piecemeal fashion, and some delay may occur between fetching operands and storing results. As a consequence, intermediate or partially completed results may be observable by other CPUs and by channel programs.

When a program interacts with the operation on another CPU, or with a channel program, the program may have to take into consideration that a single operation may consist in a series of storage references, that a storage reference may in turn consist in a series of accesses, and that the conceptual and observed sequences of these accesses may differ.
Storage references associated with instruction execution are of the following types: instruction fetches, ART-table and DAT-table fetches, and storage-operand references. For the purpose of describing the sequence of storage references, accesses to storage in order to perform ASN translation, PC-number translation, tracing, and the linkage-stack stacking and unstacking processes are considered to be storage-operand references.

Programming Note: The sequence of execution of a CPU may differ from the simple conceptual definition in the following ways:

- As observed by the CPU itself, instructions may appear to be prefetched in the access-register or home-space mode regardless of whether the mode exists at the time of the conceptual store or during the execution of the prefetched instruction. They may also appear to be prefetched when different effective addresses are used. (See "Interlocks for Virtual-Storage References."

- As observed by other CPUs and by channel programs, the execution of an instruction may appear to be performed as a sequence of piecemeal steps. This is described for each type of storage reference in the following sections.

- As observed by other CPUs and by channel programs, the storage-operand accesses associated with one instruction are not necessarily performed in the conceptual sequence. (See "Relation between Operand Accesses" on page 5-101.)

- As observed by channel programs, in certain unusual situations, the contents of storage may appear to change and then be restored to the original value. (See "Storage Change and Restoration for DAT-Associated Access Exceptions" on page 5-23)

Interlocks for Virtual-Storage References

As described in the immediately preceding sections, CPU operation appears, with certain exceptions, to be performed sequentially as observed by the CPU itself; the stores performed by one instruction generally appear to be completed before the next instruction and its operands are fetched. This appearance is maintained in overlapped machines by means of interlock circuitry that detects accesses to a common storage location.

For those instructions which alter the contents of storage and have more than one operand, the instruction definition normally describes the results that are obtained when the operands overlap in storage, this definition being in terms of a sequence of stores and fetches. The interlock circuitry is used in determining whether operand overlap exists.

The purpose of this section is to define those cases in which the machine must appear to operate sequentially, and in which operands of a single instruction must or must not be treated as overlapping.

Proper operation is provided in part by comparing effective addresses. For the purpose of this definition, the term "effective address" means an address before translation, if any, regardless of whether the address is virtual, real, or absolute. If two effective addresses have the same value, the effective addresses are said to be the same even though one may be real or in a different address space.

The values of two virtual effective addresses do not necessarily indicate whether or not the addresses designate the same storage location. The address-translation tables may be set up so that different effective addresses map to the same real address, or so that the same effective address in different address spaces maps to different real addresses.

The interlocks for virtual-storage references are considered in two situations: storage references of one instruction as they affect storage references of another instruction, and multiple storage references of a single instruction.

Interlocks between Instructions

As observed by the CPU itself, the storage accesses for operands for each instruction appear to occur in the conceptual sequence independent of the effective address used. That is, the operand stores for one instruction appear to be completed before the operand fetches for the next instruction occur. For instruction fetches, the operand stores for one instruction necessarily appear to be completed before the next instruction.
is fetched only when the same effective address is used for the operand store and the instruction fetch, and then only in the real mode, primary-space mode, or secondary-space mode.

When an instruction changes the contents of a main-storage location in which a conceptually subsequent instruction is to be executed, either directly or by means of EXECUTE, and when different effective addresses are used to designate that location for storing the result and fetching the instruction, the instruction may appear to be fetched before the store occurs. When either the storing instruction or the subsequent instruction is executed in the access-register mode or home-space mode, changes to the contents of storage are not necessarily recognized even if the effective address used to store the value and the effective address used to fetch the instruction are the same. If an intervening operation causes the prefetched instructions to be discarded, then the updated value is recognized. A definition of when prefetched instructions must be discarded is included in "Instruction Fetching" on page 5-93.

Any change to the storage key appears to be completed before the conceptually following reference to the associated storage block is made, regardless of whether the reference to the storage location is made by means of a virtual, real, or absolute address. Analogously, any conceptually prior references to the storage block appear to be completed when the key for that block is changed or inspected.

Interlocks within a Single Instruction
For those instructions which alter the contents of storage and have more than one operand, the instruction definition normally describes the results which are obtained when the operands overlap in storage. This result is normally defined in terms of the sequence of the storage accesses; that is, a portion of the results of a store-type operand must appear to be placed in storage before some portion of the other operand is fetched. This definition applies provided that the store and fetch accesses are specified by means of the same effective addresses and the same effective space designations.

When multiple address spaces are involved in the access-register mode, the term “effective space designation” is used to denote the value used by the machine to determine whether two spaces are the same. In the access-register mode, the 32-bit access-list-entry-token (ALET) value associated with each storage-operand address is called the effective space designation. When a B field of zero is specified, a value of all zeros is used for the effective space designation. If the effective space designations are different, the spaces are considered to be different even if both ALETs map to the same address-space-control-element value.

When the store and the fetch accesses are specified by means of different effective space designations or by means of different effective addresses, the operand fetch may appear to precede the operand store.

Figure 5-13 on page 5-92 summarizes the cases of overlap and the specified results, including when MOVE LONG (MVCL) sets condition code 3, for each case.

Effective space designations may be represented by ALB entries, and the test for whether two effective space designations are the same may be performed by comparing ALB entries. If the program changes an attached and valid ART-table entry without subsequently causing the execution of PURGE ALB or a COMPARE AND SWAP AND PURGE instruction that purges the ALB, two effective space designations that are the same may have different representations in the ALB, and failure to recognize operand overlap may result. The use of the ALB never causes overlap to be recognized when the effective space designations are different.

Programming Note: A single main-storage location can be accessed by means of more than one address in several ways:

1. The DAT tables may be set up such that multiple addresses in a single address space, or addresses in different address spaces, including the real address specified by a real-space designation, map to a single real address.

2. The translation of logical, instruction, and virtual addresses may be changed by loading the DAT parameters in the control registers, by changing the address-space-control bits in the PSW, or, for logical and instruction addresses, by turning DAT on or off.

3. In the access-register mode, different address spaces may be selected by means of each
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No.</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>*</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes.</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No.</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Unp.</td>
</tr>
<tr>
<td>Explanation:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* This case cannot occur.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unp. It is unpredictable whether or not the overlap is recognized.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-13. Virtual-Storage Interlocks within a Single Instruction

The primary purpose of this section on interlocks is to describe the effects caused in cases 1, 3, and 4, above.

For case 2, no effect is observable because prefetched instructions are discarded when the translation parameters are changed, and the delay of stores by a CPU is not observable by the CPU itself.

For case 5, for those instructions which fetch by using real addresses (for example, LOAD REAL ADDRESS, which fetches a segment-table entry and a page-table entry, and may fetch a region-table entry), no effect is observable because only operand accesses between instructions are involved. All instructions that store by using a real address, except STORE USING REAL ADDRESS, or that store across address spaces, except in the access-register mode, cause prefetched instructions to be discarded, and no effect is observable.

Cases 6 and 7 are situations which are defined to cause serialization, with the result that prefetched instructions are discarded. In these cases, no effect is observable.

The handling of cases 8 and 9 involves accesses as observed by other CPUs and by channel programs and is covered in the following sections in this chapter.
Instruction Fetching

Instruction fetching consists in fetching the one, two, or three halfwords designated by the instruction address in the current PSW. The immediate field of an instruction is accessed as part of an instruction fetch. If, however, an instruction designates a storage operand at the location occupied by the instruction itself, the location is accessed both as an instruction and as a storage operand. The fetch of the target instruction of EXECUTE is considered to be an instruction fetch.

The bytes of an instruction may be fetched piecemeal and are not necessarily accessed in a left-to-right direction. The instruction may be fetched multiple times for a single execution; for example, it may be fetched for testing the addressability of operands or for inspection of PER events, and it may be refetched for actual execution.

Instructions are not necessarily fetched in the sequence in which they are conceptually executed and are not necessarily fetched each time they are executed. In particular, the fetching of an instruction may precede the storage-operand references for an instruction that is conceptually earlier. The instruction fetch occurs prior to all storage-operand references for all instructions that are conceptually later.

An instruction may be prefetched by using a virtual address only when the associated DAT table entries are attached and valid or when entries which qualify for substitution for the table entries exist in the TLB. An instruction that has been prefetched may be interpreted for execution only for the same virtual address for which the instruction was prefetched.

No limit is established on the number of instructions which may be prefetched, and multiple copies of the contents of a single storage location may be fetched. As a result, the instruction executed is not necessarily the most recently fetched copy. Storing caused by other CPUs and by channel programs does not necessarily change the copy of prefetched instructions. However, if a store that is conceptually earlier is made by the same CPU using the same effective address as that by which the instruction is subsequently fetched, and the CPU is in any of the real, primary-space, and secondary-space modes when the storing instruction is executed and is in any of those modes when the subsequent instruction is executed, the updated information is obtained, except as noted below. If the effective addresses are different or if the CPU is in the access-register mode or home-space mode during either the storing execution or the execution of the instruction that is the destination of the store, the updated information is not necessarily obtained. However, the updated information is obtained if either execution is in the real mode since prefetched instructions are discarded if DAT is turned on or off.

All copies of prefetched instructions are discarded when:

- A serializing function is performed. However, for PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, SET SECONDARY ASN WITH INSTANCE, and TRACE, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store; for PROGRAM CALL and PROGRAM RETURN it is unpredictable whether or not a store into a trace-table entry or linkage-stack entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.
- The CPU enters the operating state.
- DAT is turned on or off.
- A change is made to a translation parameter in control register 1 when in the primary-space, secondary-space, or access-register mode, or in control register 13 when in the home-space mode.

The SET ADDRESS SPACE CONTROL instruction can change the translation mode between any of the primary-space, secondary-space, access-register, and home-space modes, and it performs serialization. The SET ADDRESS SPACE CONTROL FAST instruction can perform the same mode changes, but it does not serialize.

Programming Notes:

1. As observed by a CPU itself, its own instruction prefetching may be apparent when different effective addresses map to a single real address or when the CPU is in the access-register or home-space mode. This is described in “Conceptual Sequence” on
2. Any means of changing PSW bits 16 and 17, except the SET ADDRESS SPACE CONTROL FAST instruction, causes serialization to be performed and prefetched instructions to be discarded. Turning DAT on or off causes prefetched instructions to be discarded. Therefore, any change of the translation mode, except a change made by SET ADDRESS SPACE CONTROL FAST, always causes prefetched instructions to be discarded.

3. The following are some effects of instruction prefetching on one CPU as observed by other CPUs and by channel programs.

It is possible for one CPU to prefetch the contents of a storage location, after which another CPU or a channel program can change the contents of that storage location and then set a flag to indicate that the change has been made. Subsequently, the first CPU can test and find the flag set, branch to the modified location, and execute the original prefetched contents.

It is possible, if another CPU or a channel program concurrently modifies the instruction, for one CPU to recognize the changes to some but not all bit positions of an instruction.

It is possible for one CPU to prefetch an instruction and subsequently, before the instruction is executed, for another CPU to change the storage key. As a result, the first CPU may appear to execute instructions from a protected storage location. However, the copy of the instructions executed is the copy prefetched before the location was protected.

ART-Table and DAT-Table Fetches

The access-register-translation (ART) table entries are access-list designations, access-list entries, ASN-second-table entries, and authority-table entries. The dynamic-address-translation (DAT) table entries are region-table entries, segment-table entries, and page-table entries. The fetching of these entries may occur as follows:

1. An ART-table entry may be prefetched into the ART-lookaside buffer (ALB) and used from the ALB without refetching from storage, until the entry is cleared by a COMPARE AND SWAP AND PURGE, PURGE ALB, or SET PREFIX instruction, by CPU reset, or by a set-architecture SIGNAL PROCESSOR order. A DAT-table entry may be prefetched into the translation-lookaside buffer (TLB) and used from the TLB without refetching from storage, until the entry is cleared by a COMPARE AND SWAP AND PURGE, Invalidate Page Table Entry, PURGE TLB, or SET PREFIX instruction, by CPU reset, or by a set-architecture SIGNAL PROCESSOR order. ART-table and DAT-table entries are not necessarily fetched in the sequence conceptually called for; they may be fetched at any time they are attached and valid, including during the execution of conceptually previous instructions.

2. The fetching of access-list designations, access-list entries, ASN-second-table entries, and DAT-table entries appears to be word concurrent as observed by other CPUs, except that the fetching of an address-space-control element from an ASN-second-table entry appears to be doubleword concurrent as observed by other CPUs. The reference to an entry may appear to access a single byte at a time as observed by channel programs.

3. The order in which the words of an access-list entry or ASN-second-table entry are fetched is unpredictable, except that the leftmost word of an entry is fetched first. However, the leftmost word of an ASN-second-table entry is not fetched when access-list-entry token 00000000 hex is translated for BRANCH IN SUBSPACE GROUP.

4. An ART-table or DAT-table entry may be fetched even after some operand references for the instruction have already occurred. The fetch may occur as late as just prior to the actual byte access requiring the ART-table or DAT-table entry.

5. An ART-table or DAT-table entry may be fetched for each use of the address, including any trial execution, and for each reference to each byte of each operand.

6. The DAT page-table-entry fetch precedes the reference to the page. When no copy of the page-table entry is in the TLB, the fetch of the
associated segment-table entry precedes the fetch of the page-table entry. When no copy of the segment-table entry is in the TLB, the fetch of the region-third-table entry, if one is required, precedes the fetch of the segment-table entry. Similarly, the fetch of a required region-second-table entry precedes the fetch of the region-first-table entry, and the fetch of a required region-first-table entry precedes the fetch of the region-second-table entry.

7. When no copy of a region-table entry or segment-table entry designated by means of an ART-obtained address-space-control element is in the TLB, the ART fetch of the ASN-second-table entry precedes the DAT region-table-entry or segment-table-entry fetch. When no copy of a required authority-table entry is in the ALB, the ART fetch of the associated ASN-second-table entry precedes the fetch of the authority-table entry. When no copy of a required ASN-second-table entry is in the ALB, the fetch of the associated access-list entry precedes the fetch of the ASN-second-table entry. When no copy of a required access-list entry is in the ALB, the fetch of the associated access-list designation precedes the fetch of the access-list entry.

**Storage-Key Accesses**

References to the storage key are handled as follows:

1. Whenever a reference to storage is made and key-controlled protection applies to the reference, the four access-control bits and the fetch-protection bit associated with the storage location are inspected concurrently and concurrently with the reference to the storage location.

2. When storing is performed by a CPU, the change bit is set to one in the associated storage key concurrently with the completion of the store access, as observed by the CPU itself. When storing is performed by a CPU or a channel program, the change bit is set to one in the associated storage key either before or after the completion of the store access, as observed by other (if the store was performed by a CPU) or all (if the store was performed by a channel program) CPUs. As observed by other or all CPUs, the change bit is set no earlier than (1) after the last serialization function performed previously by the CPU or channel program performing the store, and (2) after the execution, by any CPU in the configuration, of the SET STORAGE KEY EXTENDED instruction that last set the associated storage key before the completion of the store. As observed by other or all CPUs, a change-bit setting necessarily occurs only when any of the following occurs subsequent to the storing operation:

   • The CPU or channel program that performed the store performs a serialization function.
   
   • The store was performed by a CPU or a channel program, and any CPU in the configuration sets the subject change bit by executing SET STORAGE KEY EXTENDED after the store access is completed. The change-bit setting due to the store access occurs before the setting by SET STORAGE KEY EXTENDED.
   
   • The store was performed by a CPU and is or will be completed, and any CPU in the configuration executes a COMPARE AND SWAP AND PURGE, INVALIDATE DAT TABLE ENTRY, or INVALIDATE PAGE TABLE ENTRY instruction that clears from the ALB or TLB of the storing CPU any entry used to complete the store. Completion of the clearing instruction is delayed until the subject store and change-bit setting have been completed.

   • The store was performed by a CPU, and that CPU examines the subject change bit by means of an INSERT STORAGE KEY EXTENDED or RESET REFERENCE BIT EXTENDED instruction. See "Relation between Storage-Key Accesses" on page 5-101.

3. The instruction SET STORAGE KEY EXTENDED causes all seven bits to be set concurrently in the storage key. The access to the storage key for SET STORAGE KEY EXTENDED follows the sequence rules for storage-operand store references and is a single-access reference.

4. The INSERT STORAGE KEY EXTENDED instruction provides a consistent image of bits 0-6 of the storage key. Similarly, the instructions INSERT VIRTUAL STORAGE KEY and TEST PROTECTION provide a con-
sistent image of bits 0-4 of the storage key. The access to the storage key for all of these instructions follows the sequence rules for storage-operand fetch references and is a single-access reference.

5. The instruction RESET REFERENCE BIT EXTENDED modifies only the reference bit. All other bits of the storage key remain unchanged. The reference bit and change bit are examined concurrently to set the condition code. The fetch and store accesses to the storage key for RESET REFERENCE BIT EXTENDED follow the sequence rules for storage-operand fetch and store references, respectively, and are single-access references.

The record of references provided by the reference bit is not necessarily accurate. However, in the majority of situations, reference recording approximately coincides with the related storage reference.

The change bit may be set in cases when no storing has occurred. See "Exceptions to Nullification and Suppression" on page 5-23.

Storage-Operand References

A storage-operand reference is the fetching or storing of the explicit operand or operands in the storage locations designated by the instruction.

During the execution of an instruction, all or some of the storage operands for that instruction may be fetched, intermediate results may be maintained for subsequent modification, and final results may be temporarily held prior to placing them in storage. Stores caused by other CPUs and by channel programs do not necessarily affect these intermediate results.

Storage-operand references are of three types: fetches, stores, and updates.

Storage-Operand Fetch References

When the bytes of a storage operand participate in the instruction execution only as a source, the operand is called a fetch-type operand, and the reference to the location is called a storage-operand fetch reference. A fetch-type operand is identified in individual instruction definitions by indicating that the access exception is for fetch.

All bits within a single byte of a fetch-type operand are accessed concurrently. When an operand consists of more than one byte, the bytes may be fetched from storage piecemeal, one byte at a time. Unless otherwise specified, the bytes are not necessarily fetched in any particular sequence.

The storage-operand fetch references of one instruction occur after those of all preceding instructions and before those of subsequent instructions, as observed by other CPUs and by channel programs. The operands of any one instruction are fetched in the sequence specified for that instruction. The CPU may fetch the operands of instructions before the instructions are executed. There is no defined limit on the length of time between when an operand is fetched and when it is used. Still, as observed by the CPU itself, its storage-operand references are performed in the conceptual sequence.

Storage-Operand Store References

When the bytes of a storage operand participate in the instruction execution only as a destination, to the extent of being replaced by the result, the operand is called a store-type operand, and the reference to the location is called a storage-operand store reference. A store-type operand is identified in individual instruction definitions by indicating that the access exception is for store.

All bits within a single byte of a store-type operand are accessed concurrently. When an operand consists of more than one byte, the bytes may be placed in storage piecemeal, one byte at a time. Unless otherwise specified, the bytes are not necessarily stored in any particular sequence.

The CPU may delay placing results in storage. There is no defined limit on the length of time that results may remain pending before they are stored. This delay does not affect the sequence in which results are placed in storage.

The results of one instruction are placed in storage after the results of all preceding instructions have been placed in storage and before any results of the succeeding instructions are stored, as observed by other CPUs and by channel programs. The results of any one instruction are stored in the sequence specified for that instruction.
The CPU does not fetch operands, ART-table entries, or DAT-table entries from a storage location until all information destined for that location by the CPU has been stored. Prefetched instructions may appear to be updated before the information appears in storage.

The stores are necessarily completed only as a result of a serializing operation and before the CPU enters the stopped state.

Storage-Operand Update References

In some instructions, the storage-operand location participates both as a source and as a destination. In these cases, the reference to the location consists first in a fetch and subsequently in a store. The operand is called an update-type operand, and the combination of the two accesses is referred to as an update reference. Instructions such as MOVE ZONES, TRANSLATE, OR (OC, OI), and ADD DECIMAL cause an update to the first-operand location. An update-type operand is identified in the individual instruction definition by indicating that the access exception is for both fetch and store.

For most instructions which have update-type operands, the fetch and store accesses associated with an update reference do not necessarily occur one immediately after the other, and it is possible for other CPUs and channel programs to make fetch and store accesses to the same location during this time. Such an update reference is sometimes called a noninterlocked-update storage reference.

For certain special instructions, the update reference is interlocked against certain accesses by other CPUs. Such an update reference is called an interlocked-update reference. The fetch and store accesses associated with an interlocked-update reference do not necessarily occur one immediately after the other, but all store accesses by other CPUs and the fetch and store accesses associated with interlocked-update references by other CPUs are prevented from occurring at the same location between the fetch and the store accesses of an interlocked-update reference. Accesses by channel programs may occur to the location during the interlock period.

The storage-operand update reference for the following instructions appears to be an interlocked-update reference as observed by other CPUs.

The instructions COMPARE AND SWAP, COMPARE AND SWAP AND PURGE, COMPARE DOUBLE AND SWAP, and TEST AND SET perform an interlocked-update reference. On models in which the STORE CHARACTERS UNDER MASK instruction with a mask of zero fetches and stores the byte designated by the second-operand address, the fetch and store accesses are an interlocked-update reference.

Within the limitations of the above requirements, the fetch and store accesses associated with an update reference follow the same rules as the fetches and stores described in the previous sections.

Programming Notes:

1. When two CPUs attempt to update information at a common main-storage location by means of a noninterlocked-update reference, it is possible for both CPUs to fetch the information and subsequently make the store access. The change made by the first CPU to store the result in such a case is lost. Similarly, if one CPU updates the contents of a field by means of a noninterlocked-update reference, but another CPU makes a store access to that field between the fetch and store parts of the update reference, the effect of the store is lost. If, instead of a store access, a CPU makes an interlocked-update reference to the common storage field between the fetch and store portions of a noninterlocked-update reference due to another CPU, any change in the contents produced by the interlocked-update reference is lost.

2. The instructions TEST AND SET, COMPARE AND SWAP, and COMPARE DOUBLE AND SWAP facilitate updating of a common storage field by two or more CPUs. To ensure that no changes are lost, all CPUs must use an instruction providing an interlocked-update reference. In addition, the program must ensure that channel programs do not store into the same storage location since such stores may occur between the fetch and store portions of an interlocked-update reference.

3. Only those bytes which are included in the result field of both operations are considered to be part of the common main-storage location. However, all bits within a common
byte are considered to be common even if the bits modified by the two operations do not overlap. As an example, if (1) one CPU executes the instruction OR (OC) with a length of 1 and the value 80 hex in the second-operand location, (2) the other CPU executes AND (NC) with a length of 1 and the value FE hex in the second-operand location, and (3) the first operand of both instructions is the same byte, then the result of one of the updates can be lost.

4. When the store access is part of an update reference by the CPU, the execution of the storing is not necessarily contingent on whether the information to be stored is different from the original contents of the location. In particular, the contents of all designated byte locations are replaced, and, for each byte in the field, the entire contents of the byte are replaced.

Depending on the model, an access to store information may be performed, for example, in the following cases:

a. Execution of the OR instruction (OI or OC) with a second operand of all zeros.
b. Execution of OR (OC) with the first-and second-operand fields coinciding.
c. For those locations of the first operand of TRANSLATE where the argument and function values are the same.

Storage-Operand Consistency

Single-Access References
A fetch reference is said to be a single-access reference if the value is fetched in a single access to each byte of the data field. In the case of overlapping operands, the location may be accessed once for each operand. A store-type reference is said to be a single-access reference if a single store access occurs to each byte location within the data field. An update reference is said to be single access if both the fetch and store accesses are each single access.

Except for the accesses associated with multiple-access references and the stores associated with storage change and restoration for DAT-associated access exceptions, all storage-operand references are single-access references.

Multiple-Access References
In some cases, multiple accesses may be made to all or some of the bytes of a storage operand. The following cases may involve multiple-access references:

1. The storage operands of the following instructions:
   - CHECKSUM
   - CIPHER MESSAGE
   - CIPHER MESSAGE WITH CHAINING
   - COMPARE AND FORM CODEWORD
   - COMPARE UNTIL SUBSTRING EQUAL
   - COMPUTE INTERMEDIATE MESSAGE DIGEST
   - COMPUTE LAST MESSAGE DIGEST
   - COMPUTE MESSAGE AUTHENTICATION CODE
   - CONVERT TO BINARY
   - CONVERT TO DECIMAL
   - CONVERT UTF-16 TO UTF-32
   - CONVERT UTF-16 TO UTF-8
   - CONVERT UTF-32 TO UTF-16
   - CONVERT UTF-8 TO UTF-16
   - CONVERT UTF-8 TO UTF-32
   - LOAD ADDRESS SPACE PARAMETERS
   - LOAD REVERSED
   - MOVE INVERSE
   - MOVE PAGE
   - MOVE WITH OFFSET
   - PACK
   - PACK ASCII
   - PACK UNICODE
   - RESUME PROGRAM
   - STORE REVERSED
   - STORE SYSTEM INFORMATION
   - TEST BLOCK
   - TRANSLATE
   - TRANSLATE EXTENDED
   - TRANSLATE ONE TO ONE
   - TRANSLATE ONE TO TWO
   - TRANSLATE TWO TO ONE
   - TRANSLATE TWO TO TWO
   - UNPACK
   - UNPACK ASCII
   - UNPACK UNICODE
   - UPDATE TREE

2. The stores into that portion of the first operand of MOVE LONG, MOVE LONG EXTENDED, or MOVE LONG UNICODE which is filled with padding bytes.
3. The storage operands of the decimal instructions.

4. The main-storage operands of PAGE IN and PAGE OUT.

5. The storage operands of the I/O instructions.

6. The stores into a trace entry.

7. The stores associated with the stop-and-store-status and store-status-at-address SIGNAL PROCESSOR orders.

8. The trap control block and trap save area used by TRAP.

9. The operands, dictionaries, and symbol-translation table of COMPRESSION CALL.

When a storage-operand store reference to a location is not a single-access reference, the value placed at a byte location is not necessarily the same for each store access; thus, intermediate results in a single-byte location may be observed by other CPUs and by channel programs.

Programming Notes:

1. When multiple fetch or store accesses are made to a single byte that is being changed by another CPU or by a channel program, the result is not necessarily limited to that which could be obtained by fetching or storing the bits individually. For example, the execution of MULTIPLY DECIMAL may consist in repetitive additions and subtractions, each of which causes the second operand to be fetched from storage and the first operand to be updated in storage.

2. When CPU instructions which make multiple-access references are used to modify storage locations being simultaneously accessed by another CPU or by a channel program, multiple store accesses to a single byte by the CPU may result in intermediate values being observed by the other CPU or by the channel program. To avoid these intermediate values (for example, when modifying a CCW chain), only instructions making single-access references should be used.

3. An instruction fetch, including the fetch of the target of EXECUTE, is a multiple-access reference.

Block-Concurrent References

For some references, the accesses to all bytes within a halfword, word, doubleword, or quadword are specified to appear to be block concurrent as observed by other CPUs. These accesses do not necessarily appear to channel programs to include more than a byte at a time. The halfword, word, doubleword, or quadword is referred to in this section as a block. When a fetch-type reference is specified to appear to be concurrent within a block, no store access to the block by another CPU is permitted during the time that bytes contained in the block are being fetched. Accesses to the bytes within the block by channel programs may occur between the fetches. When a store-type reference is specified to appear to be concurrent within a block, no access to the block, either fetch or store, is permitted by another CPU during the time that the bytes within the block are being stored. Accesses to the bytes in the block by channel programs may occur between the stores.

Consistency Specification

For all instructions in the S, RX, RXE, or RXY format, with the exception of CONVERT TO DECIMAL, CONVERT TO BINARY, LOAD PSW EXTENDED, LOAD REVERSED, RESUME PROGRAM, STORE CLOCK EXTENDED, STORE REVERSED, STORE SYSTEM INFORMATION, TRAP, and the I/O instructions, when the operand is addressed on a boundary which is integral to the size of the operand, the storage-operand references appear to be block concurrent as observed by other CPUs. For LOAD PSW EXTENDED, the accesses to each of the two doublewords of the storage operand appear to be doubleword concurrent as observed by other CPUs.

For the instructions COMPARE AND SWAP, COMPARE AND SWAP AND PURGE, and COMPARE DOUBLE AND SWAP, all accesses to the storage operand appear to be block concurrent as observed by other CPUs.

For the instruction PERFORM LOCKED OPERATION, the accesses to the even-numbered storage operands appear to be word concurrent, as observed by other CPUs, for function codes that are a multiple of 4 and appear to be doubleword concurrent, as observed by other CPUs, for function codes that are one, 2, or 3 more than a multiple of 4. The accesses to the doublewords in the parameter list appear to be doubleword concur-
rent, as observed by other CPUs, regardless of the function code.

The instructions LOAD MULTIPLE, LOAD MULTIPLE DISJOINT, LOAD MULTIPLE HIGH, STORE MULTIPLE, and STORE MULTIPLE HIGH, when the operand or operands start on a word boundary, and the instructions COMPARE LOGICAL (CLC), COMPARE LOGICAL CHARACTERS UNDER MASK, INSERT CHARACTERS UNDER MASK, LOAD CONTROL (LCTLG), STORE CHARACTERS UNDER MASK, and STORE CONTROL (STCTG) access their storage operands in a left-to-right direction, and all bytes accessed within each doubleword appear to be accessed concurrently as observed by other CPUs.

The instructions LOAD ACCESS MULTIPLE, LOAD CONTROL (LCTL), STORE ACCESS MULTIPLE, and STORE CONTROL (STCTL) access the storage operand in a left-to-right direction, and all bytes accessed within each word appear to be accessed concurrently as observed by other CPUs.

When destructive overlap does not exist, the operands of MOVE (MVC), MOVE WITH KEY, MOVE TO PRIMARY, and MOVE TO SECONDARY are accessed as follows:

1. The first operand is accessed in a left-to-right direction, and all bytes accessed within a doubleword appear to be accessed concurrently as observed by other CPUs.

2. The second operand is accessed in a left-to-right direction, and all bytes within a doubleword in the second operand that are moved into a single doubleword in the first operand appear to be fetched concurrently as observed by other CPUs. Thus, if the first and second operands begin on the same byte offset within a doubleword, the fetch of the second operand appears to be doubleword concurrent as observed by other CPUs. If the offsets within a doubleword differ by 4, the fetch of the second operand appears to be word concurrent as observed by other CPUs.

Destructive overlap is said to exist when the result location is used as a source after the result has been stored, assuming processing to be performed one byte at a time.

The operands of MOVE WITH SOURCE KEY, MOVE WITH DESTINATION KEY, and MOVE STRING are accessed the same as those of MOVE (MVC), except that destructive overlap is assumed not to exist.

Except as noted in the individual instruction descriptions, accesses to operands of MOVE LONG, MOVE LONG EXTENDED, and MOVE LONG UNICODE do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs. The operands of these instructions do appear to be accessed doubleword concurrent, as observed by other CPUs, when all of the following are true:

- Both operands start on doubleword boundaries and are an integral number of doublewords in length.
- The operands do not overlap.
- The nonpadding part of the operation is being executed.

The operands of COMPARE LOGICAL LONG, COMPARE LOGICAL LONG EXTENDED, and COMPARE LOGICAL LONG UNICODE appear to be accessed doubleword concurrent, as observed by other CPUs, when both operands start on doubleword boundaries and are an integral number of doublewords in length.

The operands of COMPARE LOGICAL STRING appear to be accessed doubleword concurrent, as observed by other CPUs, when both operands start on doubleword boundaries. The operand of SEARCH STRING appears to be accessed doubleword concurrent, as observed by other CPUs, when it starts on a doubleword boundary.

For EXCLUSIVE OR (XC), the operands are processed in a left-to-right direction, and, when the first and second operands coincide, all bytes accessed within a doubleword appear to be accessed concurrently as observed by other CPUs.

**Programming Note:** In the case of EXCLUSIVE OR (XC) designating operands which coincide exactly, the bytes within the field may appear to be accessed as many as three times, by two fetches and one store: once as the fetch portion of the first operand update, once as the second-operand fetch, and then once as the store portion of the first-operand update. Each of the three accesses appears to be doubleword concurrent as
observed by other CPUs, but the three accesses
do not necessarily appear to occur one imme-
diately after the other. One or both fetch
accesses may be omitted since the instruction can
be completed without fetching the operands.

Relation between Operand
Accesses

As observed by other CPUs and by channel pro-
grams, storage-operand fetches associated with
one instruction execution appear to precede all
storage-operand references for conceptually sub-
sequent instructions. A storage-operand store
specified by one instruction appears to precede all
storage-operand stores specified by conceptually sub-
sequent instructions, but it does not neces-
sarily precede storage-operand fetches specified
by conceptually subsequent instructions. However, a storage-operand store appears to
precede a conceptually subsequent storage-
operand fetch from the same main-storage
location.

When an instruction has two storage operands
both of which cause fetch references, it is unpredict-
table which operand is fetched first, or how
much of one operand is fetched before the other
operand is fetched. When the two operands
overlap, the common locations may be fetched
independently for each operand.

When an instruction has two storage operands the
first of which causes a store and the second a
fetch reference, it is unpredictable how much of
the second operand is fetched before the results
are stored. In the case of destructively overlap-
ning operands, the portion of the second operand
which is common to the first is not necessarily
fetched from storage.

When an instruction has two storage operands the
first of which causes an update reference and the
second a fetch reference, it is unpredictable which
operand is fetched first, or how much of one
operand is fetched before the other operand is
fetched. Similarly, it is unpredictable how much of
the result is processed before it is returned to
storage. In the case of destructively overlapping
operands, the portion of the second operand
which is common to the first is not necessarily
fetched from storage.

The independent fetching of a single location for
each of two operands may affect the program exe-
cution in the following situation. When the same
storage location is designated by two operand
addresses of an instruction, and another CPU or a
channel program causes the contents of the
location to change during execution of the instruc-
tion, the old and new values of the location may
be used simultaneously. For example, compar-
ison of a field to itself may yield a result other than
equal, or EXCLUSIVE-ORing of a field with itself
may yield a result other than zero.

Other Storage References

The restart, program, supervisor-call, external,
input/output, and machine-check PSWs appear to
be accessed doubleword concurrent as observed
by other CPUs. These references appear to occur
after the conceptually previous unit of operation
and before the conceptually subsequent unit of
operation. The relationship between the
new-PSW fetch, the old-PSW store, and the
interruption-code store is unpredictable.

Store accesses for interruption codes are not nec-
essarily single-access stores. The store accesses
for the external and supervisor-call-interruption
codes appear to occur between the conceptually
previous and conceptually subsequent operations.
The store accesses for the program-interruption
codes may precede the storage-operand refer-
ences associated with the instruction which results
in the program interruption.

Relation between Storage-Key
Accesses

As observed by other CPUs, storage-key fetches
and stores due to instructions that explicitly manip-
ulate a storage key (INSERT STORAGE KEY
EXTENDED, INSERT VIRTUAL STORAGE KEY,
RESET REFERENCE BIT EXTENDED, and SET
STORAGE KEY EXTENDED) are ordered among
themselves and among storage-operand refer-
ences as if the storage-key accesses were them-
selves storage-operand fetches and stores,
respectively.

Accesses of the access-control and fetch-
protection bits due to storage-operand references
are concurrent with the references. Accesses of
the reference and change bits due to storage-
operand references are in no particular order within the interval in which they are defined to occur. (See the description of when the change bit is set in “Storage-Key Accesses” on page 5-95.) However, whether due to an instruction that explicitly manipulates a storage key or due to a storage-operand reference, a storage-key store appears to precede a conceptually subsequent storage-key fetch from the same storage key.

Serialization

The sequence of functions performed by a CPU is normally independent of the functions performed by other CPUs and by channel programs. Similarly, the sequence of functions performed by a channel program is normally independent of the functions performed by other channel programs and by CPUs. However, at certain points in its execution, serialization of the CPU occurs. Serialization also occurs at certain points for channel programs.

CPU Serialization

All interruptions and the execution of certain instructions cause a serialization of CPU operations. A serialization operation consists in completing all conceptually previous storage accesses and related reference-bit and change-bit settings by the CPU, as observed by other CPUs and by channel programs, before the conceptually subsequent storage accesses and related reference-bit and change-bit settings occur. Serialization affects the sequence of all CPU accesses to storage and to the storage keys, except for those associated with ART-table-entry and DAT-table-entry fetching.

Serialization is performed by CPU reset, all interruptions, and by the execution of the following instructions:

- The general instructions BRANCH ON CONDITION (BCR) with the M₁ and R₂ field containing all ones and all zeros, respectively, and COMPARE AND SWAP, COMPARE DOUBLE AND SWAP, STORE CLOCK, STORE CLOCK EXTENDED, SUPERVISOR CALL, and TEST AND SET.
- LOAD PSW, LOAD PSW EXTENDED, and SET STORAGE KEY EXTENDED.
- All I/O instructions.
- PURGE ALB, PURGE TLB, and SET PREFIX. PURGE ALB and SET PREFIX also cause the ALB to be cleared of all entries. PURGE TLB and SET PREFIX also cause the TLB to be cleared of all entries.
- SIGNAL PROCESSOR. The set-architecture SIGNAL PROCESSOR order causes serialization on all CPUs in the configuration.
- INVALIDATE PAGE TABLE ENTRY.
- TEST BLOCK.
- MOVE TO PRIMARY, MOVE TO SECONDARY, and SET ADDRESS SPACE CONTROL.
- PROGRAM CALL, and, when the state entry to be unstacked is a program-call state entry, PROGRAM RETURN. However, it is unpredictable whether or not a store into a trace-table entry or linkage-stack entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.
- PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, and SET SECONDARY ASN WITH INSTANCE, and TRACE. However, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.
- PERFORM LOCKED OPERATION. Serialization is performed immediately after the lock is obtained and again immediately before it is released. However, values fetched from the parameter list before the lock is obtained are not necessarily refetched.
- PAGE IN and PAGE OUT.
- COMPARE AND SWAP AND PURGE, which can also cause the ALB and the TLB to be cleared of all entries on all CPUs

The four trace functions — branch tracing, ASN tracing, mode tracing, and explicit tracing — cause serialization to be performed before the trace action and after completion of the trace action. However, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.
The sequence of events associated with a serializing operation is as follows:

1. All conceptually previous storage accesses by the CPU are completed as observed by other CPUs and by channel programs. This includes all conceptually previous stores and changes to the storage keys.

2. The normal function associated with the serializing operation is performed. In the case of instruction execution, operands are fetched, and the storing of results is completed. The exceptions are LOAD PSW, LOAD PSW EXTENDED, and SET PREFIX, in which the operand may be fetched before previous stores have been completed, and interruptions, in which the interruption code and associated fields may be stored prior to the serialization. The fetching of the serializing instruction occurs before the execution of the instruction and may precede the execution of previous instructions, but may not precede the completion of any previous serializing operation. In the case of an interruption, the old PSW, the interruption code, and other information, if any, are stored, and the new PSW is fetched, but not necessarily in that sequence.

3. Finally, instruction fetch and operand accesses for conceptually subsequent operations may begin.

A serializing function affects the sequence of storage accesses that are under the control of the CPU in which the serializing function takes place. It does not affect the sequence of storage accesses under the control of other CPUs and of channel programs.

Programming Notes:

1. The following are some effects of a serializing operation:
   a. When the execution of an instruction changes the contents of a storage location that is used as a source of a following instruction and when different addresses are used to designate the same absolute location for storing the result and fetching the instruction, a serializing operation following the change ensures that the modified instruction is executed.
   b. When a serializing operation takes place, other CPUs and channel programs observe instruction and operand fetching and result storing to take place in the sequence established by the serializing operation.

2. Storing into a location from which a serializing instruction is fetched does not necessarily affect the execution of the serializing instruction unless a serializing function has been performed after the storing and before the execution of the serializing instruction.

3. Following is an example showing the effects of serialization. Location A initially contains X'FF'.

   CPU 1             CPU 2
   MVI A,X'00'      G CLI A,X'00'
   BCR 15,0         BNE G

   The BCR 15,0 instruction executed by CPU 1 is a serializing instruction that ensures that the store by CPU 1 at location A is completed. However, CPU 2 may loop indefinitely, or until the next I/O or external interruption on CPU 2, because CPU 2 may already have fetched from location A for every execution of the CLI instruction. A serializing instruction must be in the CPU-2 loop to ensure that CPU 2 will again fetch from location A.

Channel-Program Serialization

Serialization of a channel program occurs as follows:

1. All storage accesses and storage-key accesses by the channel program follow initiation of the execution of START SUBCHANNEL, or, if suspended, RESUME SUBCHANNEL, as observed by CPUs and by other channel programs. This includes all accesses for the CCWs, IDAWs, and data.

2. All storage accesses and storage-key accesses by the channel program are completed, as observed by CPUs and by other channel programs, before the subchannel status indicating status-pending with primary status is made available to any CPU.

3. If a CCW contains a PCI flag or a suspend flag which is one, all storage accesses and storage-key accesses due to CCWs preceding it in the CCW chain are completed, as
observed by CPUs and by other channel programs, before the subchannel status indicating status-pending with intermediate status (PCI or suspended) is made available to any CPU.

The serialization of a channel program does not affect the sequence of storage accesses or storage-key accesses performed by other channel programs or by a CPU.
Chapter 6. Interruptions

The interruption mechanism permits the CPU to change its state as a result of conditions external to the configuration, within the configuration, or within the CPU itself. To permit fast response to conditions of high priority and immediate recognition of the type of condition, interruption conditions are grouped into six classes: external,
input/output, machine check, program, restart, and supervisor call.

**Interruption Action**

An interruption consists in storing the current PSW as an old PSW, storing information identifying the cause of the interruption, and fetching a new PSW. Processing resumes as specified by the new PSW.

The old PSW stored on an interruption normally contains the address of the instruction that would have been executed next had the interruption not occurred, thus permitting resumption of the interrupted program. For program and supervisor-call interruptions, the information stored also contains a code that identifies the length of the last-executed instruction, thus permitting the program to respond to the cause of the interruption. In the case of some program conditions for which the normal response is reexecution of the instruction causing the interruption, the instruction address directly identifies the instruction last executed.

Except for restart, an interruption can occur only when the CPU is in the operating state. The restart interruption can occur with the CPU in either the stopped or operating state.

The details of source identification, location determination, and instruction execution are explained in later sections and are summarized in Figure 6-1 on page 6-3.
<table>
<thead>
<tr>
<th>Source Identification</th>
<th>Interruption Code</th>
<th>PSW-Mask Bits</th>
<th>Mask Bits in Ctrl Registers</th>
<th>ILC Set</th>
<th>Execution of Instruction Identified by Old PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE CHECK</td>
<td>Locations 232-239</td>
<td>13</td>
<td>u</td>
<td></td>
<td>terminated or nullified</td>
</tr>
<tr>
<td>(old PSW 352,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unaffected</td>
</tr>
<tr>
<td>new PSW 480)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exigent condition</td>
<td></td>
<td>13</td>
<td>14, 35-39</td>
<td>u</td>
<td>nullified</td>
</tr>
<tr>
<td>Repressible cond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>suppressed</td>
</tr>
<tr>
<td>SUPERVISOR CALL</td>
<td>Locations 138-139</td>
<td></td>
<td></td>
<td></td>
<td>completed</td>
</tr>
<tr>
<td>(old PSW 320,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new PSW 448)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction bits</td>
<td>000000000 sssssss</td>
<td>1,2</td>
<td></td>
<td></td>
<td>completed</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Locations 142-143</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(old PSW 336,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new PSW 464)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>000000000 p00000100p000001000001</td>
<td>1,2,3</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Privileged oper</td>
<td>000000000 p000000010000000100000001</td>
<td>2,3</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>000000000 p0000011100000000000111</td>
<td>2</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protection</td>
<td>000000000 p0000100000000000100000</td>
<td>1,2,3</td>
<td>terminated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressing</td>
<td>000000000 p0000101000000000000101</td>
<td>1,2,3</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specification</td>
<td>000000000 p0000110000000000110001</td>
<td>0,1,2,3</td>
<td>completed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>000000000 p0000111000000000111001</td>
<td>1,2,3</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-pt overflow</td>
<td>000000000 p0001010000010010000100</td>
<td>1,2,3</td>
<td>completed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-point divide</td>
<td>000000000 p00010010000010100001001</td>
<td>1,2,3</td>
<td>completed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal overflow</td>
<td>000000000 p00010100000100100001000</td>
<td>1,2,3</td>
<td>completed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal divide</td>
<td>000000000 p00010011000010110001011</td>
<td>2,3</td>
<td>completed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFP exp. overflow</td>
<td>000000000 p00011000000110000001100</td>
<td>1,2,3</td>
<td>completed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFP exp. underflow</td>
<td>000000000 p00011010000110100001101</td>
<td>22</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFP significance</td>
<td>000000000 p00011100000111000001111</td>
<td>23</td>
<td>completed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFP divide</td>
<td>000000000 p00011111000111110001111</td>
<td>00F</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment translation</td>
<td>000000000 p00100000000100000100000</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page translation</td>
<td>000000000 p00100001000100010001001</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translation spec</td>
<td>000000000 p00100100000100100001001</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special operation</td>
<td>000000000 p00100110000100100001011</td>
<td>0013, 0, 33</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand</td>
<td>000000000 p00101010000100100001011</td>
<td>0015</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace table</td>
<td>000000000 p00101100000100100001011</td>
<td>0016</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space-switch event</td>
<td>000000000 p00111000000110000001100</td>
<td>001C</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HFP square root</td>
<td>000000000 p00111010000110100001101</td>
<td>001D</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC-transl spec</td>
<td>000000000 p00111111000111110001111</td>
<td>001F</td>
<td>suppressed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-1 (Part 1 of 3). Interruption Action
<table>
<thead>
<tr>
<th>Source Identification</th>
<th>Interruption Code</th>
<th>PSW-Mask Bits in Ctrl Registers</th>
<th>Mask Bits Reg, Bit</th>
<th>ILC Set</th>
<th>Execution of Instruction Identified by Old PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFX translation</td>
<td>00000000 01010000</td>
<td>0020</td>
<td>1,2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>ASX translation</td>
<td>00000000 01010001</td>
<td>0021</td>
<td>1,2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>LX translation</td>
<td>00000000 01010010</td>
<td>0022</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>EX translation</td>
<td>00000000 01010011</td>
<td>0023</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Primary authority</td>
<td>00000000 01010100</td>
<td>0024</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Secondary auth</td>
<td>00000000 01010101</td>
<td>0025</td>
<td>1,2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>LX translation</td>
<td>00000000 01010100</td>
<td>0026</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>LSF translation</td>
<td>00000000 01010111</td>
<td>0027</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>ALET specification</td>
<td>00000000 01010110</td>
<td>0028</td>
<td>1,2,3</td>
<td>suppressed</td>
<td></td>
</tr>
<tr>
<td>ALEN translation</td>
<td>00000000 01010101</td>
<td>0029</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>ALE sequence</td>
<td>00000000 01010100</td>
<td>002A</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>ASTE validity</td>
<td>00000000 01010111</td>
<td>002B</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>ASTE sequence</td>
<td>00000000 01011000</td>
<td>002C</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Extended authority</td>
<td>00000000 01011010</td>
<td>002D</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>LSTE sequence</td>
<td>00000000 01011110</td>
<td>002E</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>ASTE instance</td>
<td>00000000 01011111</td>
<td>002F</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Stack full</td>
<td>00000000 01100000</td>
<td>0030</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Stack empty</td>
<td>00000000 01100001</td>
<td>0031</td>
<td>1,2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Stack specification</td>
<td>00000000 01100100</td>
<td>0032</td>
<td>1,2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Stack type</td>
<td>00000000 01100110</td>
<td>0033</td>
<td>1,2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Stack operation</td>
<td>00000000 01101000</td>
<td>0034</td>
<td>1,2</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>ASCE type</td>
<td>00000000 01110000</td>
<td>0038</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Region first trans</td>
<td>00000000 01111000</td>
<td>0039</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Region second trans</td>
<td>00000000 01111010</td>
<td>003A</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Region third trans</td>
<td>00000000 01111100</td>
<td>003B</td>
<td>1,2,3</td>
<td>nullified</td>
<td></td>
</tr>
<tr>
<td>Monitor event</td>
<td>00000000 01000000</td>
<td>0040</td>
<td>8, 32-47</td>
<td>completed</td>
<td></td>
</tr>
<tr>
<td>PER event</td>
<td>00000000 1nnnnnnn</td>
<td>0080</td>
<td>9, 32-46</td>
<td>completed</td>
<td></td>
</tr>
<tr>
<td>Crypto operation</td>
<td>00000000 01011001</td>
<td>0019</td>
<td>2</td>
<td>nullified</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6-1 (Part 2 of 3). Interruption Action**
<table>
<thead>
<tr>
<th>Source Identification</th>
<th>Interruption Code</th>
<th>PSW-Mask Bits</th>
<th>Mask Bits in Ctrl Registers</th>
<th>ILC Set</th>
<th>Execution of Instruction Identified by Old PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT/OUTPUT</td>
<td>Locations 184-191</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(old PSW 368, new PSW 496)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O-interruption subclass</td>
<td></td>
<td>6</td>
<td>6, 32-39</td>
<td>u</td>
<td>unaffected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESTART</td>
<td>None</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(old PSW 288, new PSW 416)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restart key</td>
<td></td>
<td></td>
<td></td>
<td>u</td>
<td>unaffected</td>
</tr>
</tbody>
</table>

**Explanation:**

Locations for the old PSWs, new PSWs, and interruption codes are real locations.

1. A model-independent machine-check interruption code of 64 bits is stored at real locations 232-239.

2. The effect of the machine-check condition is indicated by bits in the machine-check-interruption code. The setting of these bits indicates the extent of the damage and whether the unit of operation is nullified, terminated, or unaffected.

3. The interruption code in the column labeled "Hex" is the hex code for the basic interruption; this code does not show the effects of concurrent interruption conditions represented by n or p in the column labeled "Binary."

4. Bits 32-39 of control register 6 provide detailed masking of I/O-interruption subclasses 0-7 respectively.

5. When the interruption code indicates a PER event, an ILC of 0 may be stored only when bits 8-15 of the interruption code are 10000110 (PER, specification).

6. The unit of operation is completed, unless a program exception concurrently indicated causes the unit of operation to be nullified, suppressed, or terminated.

n A possible nonzero code indicating another concurrent program-interruption condition

p If one, the bit indicates a concurrent PER-event interruption condition.

s Bits of the I field of SUPERVISOR CALL.

u Not stored.

---

**Figure 6-1 (Part 3 of 3). Interruption Action**

### Interruption Code

The six classes of interruptions (external, I/O, machine check, program, restart, and supervisor call) are distinguished by the storage locations at which the old PSW is stored and from which the new PSW is fetched. For most classes, the causes are further identified by an interruption code and, for some classes, by additional information placed in permanently assigned real storage locations during the interruption. (See also "Assigned Storage Locations" on page 3-58.) For external, program, and supervisor-call interruptions, the interruption code consists of 16 bits. For external interruptions, the interruption code is stored at real locations 134-135. A parameter may be stored at real locations 128-131, or a CPU address may be stored at real locations 132-133.

For I/O interruptions, the I/O-interruption code is stored at real locations 184-191. The I/O-interruption code consists of a 32-bit subsystem-identification word and a 32-bit interruption parameter.

For machine-check interruptions, the interruption code consists of 64 bits and is stored at real locations 232-239. Additional information for identifying the cause of the interruption and for recovering the state of the machine may be provided by the contents of the machine-check failing-storage...
address and the contents of the fixed-logout and machine-check-save areas. (See Chapter 11, "Machine-Check Handling.")

For program interruptions, the interruption code is stored at real locations 142-143, and the instruction-length code is stored in bit positions 5 and 6 of real location 141. Further information may be provided in the form of the data-exception code (DXC), monitor-class number, PER code, addressing-and-translation-mode identification, PER address, exception access identification, PER access identification, operand-access identification, translation-exception identification, and monitor code, which are stored at real locations 144-162 and 168-183.

Enabling and Disabling

By means of mask bits in the current PSW, floating-point-control (FPC) register, and control registers, the CPU may be enabled or disabled for all external, I/O, and machine-check interruptions and for some program interruptions. When a mask bit is one, the CPU is enabled for the corresponding class of interruptions, and those interruptions can occur.

When a mask bit is zero, the CPU is disabled for the corresponding interruptions. The conditions that cause I/O interruptions remain pending. External-interruption conditions either remain pending or persist until the cause is removed. Machine-check-interruption conditions, depending on the type, are ignored, remain pending, or cause the CPU to enter the check-stop state. The disallowed program-interruption conditions are ignored, except that some causes are indicated also by the setting of the condition code, and IEEE exceptions set flags in the FPC register. The setting of the HFP-significance and HFP-exponent-underflow program-mask bits affects the manner in which HFP operations are completed when the corresponding condition occurs. Similarly, the setting of the IEEE mask bits in the FPC register affects the manner in which BFP operations are completed when the corresponding condition occurs.

Programming Notes:

1. Mask bits in the PSW provide a means of disallowing most maskable interruptions; thus, subsequent interruptions can be disallowed by the new PSW introduced by an interruption. Furthermore, the mask bits can be used to establish a hierarchy of interruption priorities, where a condition in one class can interrupt the program handling a condition in another class but not vice versa. To prevent an interruption-handling routine from being interrupted before the necessary housekeeping steps are performed, the new PSW must disable the CPU for further interruptions within the same class or within a class of lower priority.

2. Because the mask bits in control registers are not changed as part of the interruption procedure, these masks cannot be used to prevent an interruption immediately after a previous interruption in the same class. The mask bits in control registers provide a means for selectively enabling the CPU for some sources and disabling it for others within the same class.

3. Controlling bits exist for several program interruptions, but with no mask bit in the PSW. Such bits include the IEEE mask bits in the FPC register, the monitor masks in bit positions 48-63 of control register 8, and the primary space-switch-event-control bit, bit 57 of control register 1. A bit of this nature is somewhat arbitrarily considered to be a “mask” bit only if the polarity is such that interruption is enabled when the bit is one. Thus, for example, the SSM-suppression-control bit, bit 33 of control register 0, is considered to be a mask bit, while the AFP-register-control bit, bit 45 of control register 0, is not. Regardless of the polarity of such control bits, to avoid another program interruption, an interruption-handling routine must avoid issuing instructions subject to these bits until they have been set appropriately.

Handling of Floating Interruption Conditions

An interruption condition which can be presented to any CPU in the configuration is called a floating interruption condition. The condition is presented to the first CPU in the configuration which is enabled for the corresponding interruption and
which can perform the interruption, and then the condition is cleared and not presented to any other CPU in the configuration. A CPU cannot perform the interruption when it is in the check-stop state, has an invalid prefix, is in a string of program interruptions due to a specification exception of the type which is recognized early or is in the stopped state. However, a CPU with the rate control set to instruction step can perform the interruption when the start key is activated.

Service signal, I/O, and certain machine-check conditions are floating interruption conditions.

Instruction-Length Code

The instruction-length code (ILC) occupies two bit positions and provides the length of the last instruction executed. It permits identifying the instruction causing the interruption when the instruction address in the old PSW designates the next sequential instruction. The ILC is provided also by the BRANCH AND LINK instructions in the 24-bit addressing mode.

The ILC for program and supervisor-call interruptions is stored in bit positions 5 and 6 of the bytes at real locations 141 and 137, respectively. For external, I/O, machine-check, and restart interruptions, the ILC is not stored since it cannot be related to the length of the last-executed instruction.

For supervisor-call and program interruptions, a nonzero ILC identifies in halfwords the length of the instruction that was last executed. That instruction may be one for which a specification exception was recognized due to an odd instruction address or for which an access exception (addressing, ASCE-type, page-translation, protection, region-translation, segment-translation, or translation-specification) was recognized during the fetching of the instruction. Whenever an instruction is executed by means of EXECUTE, instruction-length code 2 is set to indicate the length of EXECUTE and not that of the target instruction.

The value of a nonzero instruction-length code is related to the leftmost two bits of the instruction. The value does not depend on whether the operation code is assigned or on whether the instruction is installed. The following table summarizes the meaning of the instruction-length code:

<table>
<thead>
<tr>
<th>ILC</th>
<th>Instr.</th>
<th>Instruction Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>Binary</td>
<td>0-1</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>Not available</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>One halfword</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Two halfwords</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Two halfwords</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>Three halfwords</td>
</tr>
</tbody>
</table>

Zero ILC

Instruction-length code 0, after a program interruption, indicates that the instruction address stored in the old PSW does not identify the instruction causing the interruption.

An ILC of 0 occurs when a specification exception due to a PSW-format error is recognized as part of early exception recognition and the PSW has been introduced by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or an interruption. (See "Exceptions Associated with the PSW" on page 6-9.) In the case of LOAD PSW, LOAD PSW EXTENDED, or PROGRAM RETURN, the instruction address of the instruction or of EXECUTE has been replaced by the instruction address in the new PSW. When the invalid PSW is introduced by an interruption, the PSW-format error cannot be attributed to an instruction.

In the case of LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, and the supervisor-call interruption, a PER event may be indicated concurrently with a specification exception for which the ILC is 0.

In the case of a PROGRAM RETURN instruction that causes both a space-switch event and a PSW-format error, the space-switch event is recognized, but it is unpredictable whether the ILC is 0 or 1, or 0 or 2 if EXECUTE was used.

ILC on Instruction-Fetching Exceptions

When a program interruption occurs because of an exception that prohibits access to the instruction, the instruction is considered to have been executed without being fetched, and the instruction-length code cannot be set on the basis of the first two bits of the instruction. As far as the significance of the ILC for this case is concerned, the following two situations are distinguished:
1. When an odd instruction address causes a specification exception to be recognized or when an addressing, protection, or translation-specification exception is encountered on fetching an instruction, the ILC is set to 1, 2, or 3, indicating the multiple of 2 by which the instruction address has been incremented. It is unpredictable whether the instruction address is incremented by 2, 4, or 6. By reducing the instruction address in the old PSW by the number of halfword locations indicated in the ILC, the instruction address originally appearing in the PSW may be obtained.

2. When an ASCE-type, region-translation, segment-translation, or page-translation exception is recognized while fetching an instruction, the ILC is arbitrarily set to 1, 2, or 3. In this case, the operation is nullified, and the instruction address is not incremented.

The ILC is not necessarily related to the first two bits of the instruction when the first halfword of an instruction can be fetched but an access exception is recognized on fetching the second or third halfword. The ILC may be arbitrarily set to 1, 2, or 3 in these cases. The instruction address is or is not updated, as described in situations 1 and 2 above.

When any exceptions are encountered on fetching the target instruction of EXECUTE, the ILC is 2.

**Programming Notes:**

1. A nonzero instruction-length code for a program interruption indicates the number of halfword locations by which the instruction address in the program old PSW must be reduced to obtain the instruction address of the last instruction executed, unless one of the following situations exists:
   a. The interruption is caused by an exception resulting in nullification.
   b. An interruption for a PER event occurs before the execution of an interruptible instruction is completed, and no other program-interruption condition is indicated concurrently.
   c. The interruption is caused by a PER event or space-switch event due to LOAD PSW, LOAD PSW EXTENDED, or a branch or linkage instruction, including SUPERVISOR CALL (but not including MONITOR CALL).
   d. The interruption is caused by an addressing exception or protection exception for the storage operand of a LOAD CONTROL instruction that loads the control register (1 or 13) containing the address-space-control element that specifies the address space from which instructions are fetched.

For situations a and b above, the instruction address in the PSW is not incremented, and the instruction designated by the instruction address is the same as the last one executed. These situations are the only ones in which the instruction address in the old PSW identifies the instruction causing the exception. Situation b can be distinguished from a PER event indicated after completion of an interruptible or noninterruptible instruction in that, for situation b, the instruction address in the PSW is the same as the PER address in the doubleword at real location 152.

For situation c, the instruction address has been replaced as part of the operation, and the address of the last instruction executed cannot be calculated using the one appearing in the program old PSW.

For situation d, the effective address of the last instruction executed can be calculated, but, since the address-space-control element for the instruction address space is unpredictable, the corresponding real address is unknown.

2. The instruction-length code (ILC) is redundant when a PER event is indicated since the PER address in the doubleword at real location 152 identifies the instruction causing the interruption (or the EXECUTE instruction, as appropriate). Similarly, the ILC is redundant when the operation is nullified, since in this case the instruction address in the PSW is not incremented. If the ILC value is required in this case, it can be derived from the operation code of the instruction identified by the old PSW.

3. The address of the last instruction executed before a program interruption is insufficient to locate the program problem if one of the following situations exists:
a. The interruption is caused by an access exception encountered in fetching an instruction, and the instruction address was introduced into the PSW by a means other than sequential operation (by a branch or linkage instruction, LOAD PSW, LOAD PSW EXTENDED, an interruption, or conclusion of an IPL sequence).

b. The interruption is caused by a specification exception due to an odd instruction address, which necessarily also results from introduction of an instruction address into the PSW.

c. The interruption is caused by an early specification exception due to a STORE THEN OR SYSTEM MASK or SET SYSTEM MASK instruction that switches to or from the real mode while introducing invalid values in bit positions 0-7 of the PSW.

For situations a and b, the instruction address was replaced by the operation preceding the last instruction execution, and the address of the program location related to that preceding operation is unavailable.

For situation c, the address of the last instruction executed is available, but the corresponding real address is unknown.

4. The address of the last instruction executed is not available when an interruption is caused by an early specification exception due to a LOAD PSW, LOAD PSW EXTENDED, or PROGRAM RETURN instruction or an interruption.

Exceptions Associated with the PSW

Exceptions associated with erroneous information in the current PSW may be recognized when the information is introduced into the PSW or may be recognized as part of the execution of the next instruction. Errors in the PSW which are specification-exception conditions are called PSW-format errors.

Early Exception Recognition

For the following error conditions, a program interruption for a specification exception occurs immediately after the PSW becomes active:

- Any of the unassigned bits (0, 2-4, 24-30, or 33-63) is a one.
- Bit 12 is a one.
- Bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros.
- Bits 31 and 32 are both zero, and bits 64-103 are not all zeros.
- Bits 31 and 32 are one and zero, respectively.

The interruption occurs regardless of whether the wait state is specified. If the invalid PSW causes the CPU to become enabled for a pending I/O, external, or machine-check interruption, the program interruption occurs instead, and the pending interruption is subject to the mask bits of the new PSW introduced by the program interruption.

When an interruption or the execution of LOAD PSW, LOAD PSW EXTENDED, or PROGRAM RETURN introduces a PSW with one of the above error conditions, the instruction-length code is set to 0, and the newly introduced PSW is stored unmodified as the old PSW. When one of the above error conditions is introduced by execution of SET SYSTEM MASK or STORE THEN OR SYSTEM MASK, the instruction-length code is set to 2, and the instruction address is incremented by 4. The PSW containing the invalid value introduced into the system-mask field is stored as the old PSW.

Late Exception Recognition

For the following conditions, the exception is recognized as part of the execution of the next instruction:

- A specification exception is recognized due to an odd instruction address in the PSW (PSW bit 127 is one).
- An access exception (addressing, ASCE-type, page-translation, protection, region-translation, segment-translation, or translation-specification) is associated with the location designated by the instruction address or with the location of the second or third halfword of
the instruction starting at the designated instruction address.

The instruction-length code and instruction address stored in the program old PSW under these conditions are discussed in "ILC on Instruction-Fetching Exceptions" on page 6-7 and an example is given in Figure 4-9 on page 4-39.

If an I/O, external, or machine-check-interruption condition is pending and the PSW causes the CPU to be enabled for that condition, the corresponding interruption occurs, and the PSW is not inspected for exceptions which are recognized late. Similarly, a PSW specifying the wait state is not inspected for exceptions which are recognized late.

Programming Notes:

1. The execution of LOAD ADDRESS SPACE PARAMETERS, LOAD PSW, LOAD PSW EXTENDED, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, RESUME PROGRAM, SET PREFIX, SET SECONDARY ASN, SET SYSTEM MASK, STORE THEN AND SYSTEM MASK, and STORE THEN OR SYSTEM MASK is suppressed on an addressing or protection exception, and hence the program old PSW provides information concerning the program causing the exception.

2. When the first halfword of an instruction can be fetched but an access exception is recognized on fetching the second or third halfword, the ILC is not necessarily related to the operation code.

3. If the new PSW introduced by an interruption contains a PSW-format error, a string of interruptions may occur. (See “Priority of Interruptions” on page 6-48)

External Interruption

The external interruption provides a means by which the CPU responds to various signals originating from either inside or outside the configuration.

An external interruption causes the old PSW to be stored at real locations 304-319 and a new PSW to be fetched from real locations 432-447.

The source of the interruption is identified in the instruction code which is stored at real locations 134-135. The instruction-length code is not stored.

Additionally, for the malfunction-alert, emergency-signal, and external-call conditions, a 16-bit CPU address is associated with the source of the interruption and is stored at real locations 132-133. When the CPU address is stored, bit 6 of the interruption code is set to one. For all other conditions, no CPU address is stored, bit 6 of the interruption code is set to zero, and zeros are stored at real locations 132-133.

For the ETR and service-signal interruptions, a 32-bit parameter is associated with the interruption and is stored at real locations 128-131. Bit 5 of the external-interruption code indicates that a parameter has been stored. When bit 5 is zero, the contents of real locations 128-131 remain unchanged.

External-interruption conditions are of two types: those for which an interruption-request condition is held pending, and those for which the condition directly requests the interruption. Clock comparator and CPU timer are conditions which directly request external interruptions. If a condition which directly requests an external interruption is removed before the request is honored, the request does not remain pending, and no interruption occurs. Conversely, the request is not cleared by the interruption, and, if the condition persists, more than one interruption may result from a single occurrence of the condition.

When several interruption requests for a single source are generated before the interruption occurs, and the interruption condition is of the type which is held pending, only one request for that source is preserved and remains pending.

An external interruption for a particular source can occur only when the CPU is enabled for interruption by that source. The external interruption occurs at the completion of a unit of operation. The external mask, PSW bit 7, and external subclass-mask bits in control register 0 control whether the CPU is enabled for a particular source. Each source for an external interruption has a subclass-mask bit assigned to it, and the source can cause an interruption only when the
external-mask bit is one and the corresponding subclass-mask bit is one.

When the CPU becomes enabled for a pending external-interruption condition, the interruption occurs at the completion of the instruction execution or interruption that causes the enabling.

More than one source may present a request for an external interruption at the same time. When the CPU becomes enabled for more than one concurrently pending request, the interruption occurs for the pending condition or conditions having the highest priority.

The priorities for external-interruption requests in descending order are as follows:

- Interrupt key
- Malfunction alert
- Emergency signal
- External call
- Clock comparator
- CPU timer
- ETR
- Service signal

All requests are honored one at a time. When more than one emergency-signal request exists at a time or when more than one malfunction-alert request exists at a time, the request associated with the smallest CPU address is honored first.

Clock Comparator

An interruption request for the clock comparator exists whenever either of the following conditions is met:

1. The TOD clock is in the set or not-set state, and the value of the clock comparator is less than the value in the compared portion of the TOD clock, both compare values being considered unsigned binary integers.
2. The TOD clock is in the error or not-operational state.

If the condition responsible for the request is removed before the request is honored, the request does not remain pending, and no interruption occurs. Conversely, the request is not cleared by the interruption, and, if the condition persists, more than one interruption may result from a single occurrence of the condition.

When the TOD clock is set or changes state, interruption conditions, if any, that are due to the clock comparator may or may not be recognized for up to 1.048576 seconds after the change.

The subclass-mask bit is in bit position 52 of control register 0. This bit is initialized to zero.

The clock-comparator condition is indicated by an external-interruption code of 1004 hex.

CPU Timer

An interruption request for the CPU timer exists whenever the CPU-timer value is negative (bit 0 of the CPU timer is one). If the value is made positive before the request is honored, the request does not remain pending, and no interruption occurs. Conversely, the request is not cleared by the interruption, and, if the condition persists, more than one interruption may occur from a single occurrence of the condition.

When the TOD clock is set or changes state, interruption conditions, if any, that are due to the CPU timer may or may not be recognized for up to 1.048576 seconds after the change.

The subclass-mask bit is in bit position 53 of control register 0. This bit is initialized to zero.

The CPU-timer condition is indicated by an external-interruption code of 1005 hex.

Emergency Signal

An interruption request for an emergency signal is generated when the CPU accepts the emergency-signal order specified by a SIGNAL PROCESSOR instruction addressing this CPU. The instruction may have been executed by this CPU or by another CPU in the configuration. The request is preserved and remains pending in the receiving CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

Facilities are provided for holding a separate emergency-signal request pending in the receiving CPU for each CPU in the configuration, including the receiving CPU itself.

The subclass-mask bit is in bit position 49 of control register 0. This bit is initialized to zero.
The emergency-signal condition is indicated by an external-interruption code of 1201 hex. The address of the CPU that executed the SIGNAL PROCESSOR instruction is stored at real locations 132-133.

**ETR**

An interruption request for the ETR is generated when a port-availability change occurs at any port in the current CPC-port group or when an ETR alert occurs. The terms specific to the ETR are not defined in this publication.

If the same ETR condition occurs more than once before the interruption occurs, the request is generated only once. The request is generated for all CPUs in the configuration.

The subclass-mask bit is in bit position 59 of control register 0. This bit is initialized to zero.

The ETR condition is indicated by an external-interruption code of 1406 hex.

**Interrupt Key**

An interruption request for the interrupt key is generated when the operator activates that key. The request is preserved and remains pending in the CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

When the interrupt key is activated while the CPU is in the load state, it depends on the model whether an interruption request is generated or the condition is lost.

The subclass-mask bit is in bit position 57 of control register 0. This bit is initialized to one.

The interrupt-key condition is indicated by an external-interruption code of 0040 hex.

**Malfunction Alert**

An interruption request for a malfunction alert is generated when another CPU in the configuration enters the check-stop state or loses power. The request is preserved and remains pending in the receiving CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

Facilities are provided for holding a separate malfunction-alert request pending in the receiving CPU for each of the other CPUs in the configuration. Removal of a CPU from the configuration does not generate a malfunction-alert condition.

The subclass-mask bit is in bit position 48 of control register 0. This bit is initialized to zero.

The malfunction-alert condition is indicated by an external-interruption code of 1200 hex. The address of the CPU that generated the condition is stored at real locations 132-133.

**External Call**

An interruption request for an external call is generated when the CPU accepts the external-call order specified by a SIGNAL PROCESSOR instruction addressing this CPU. The instruction may have been executed by this CPU or by another CPU in the configuration. The request is preserved and remains pending in the receiving CPU until it is cleared. The pending request is cleared when it causes an interruption and by CPU reset.

Only one external-call request, along with the processor address, may be held pending in a CPU at a time.

The subclass-mask bit is in bit position 50 of control register 0. This bit is initialized to zero.

The external-call condition is indicated by an external-interruption code of 1202 hex. The address of the CPU that executed the SIGNAL PROCESSOR instruction is stored at real locations 132-133.

**Service Signal**

An interruption request for a service signal is generated upon the completion of certain configuration-control and maintenance functions, such as those initiated by means of the model-dependent DIAGNOSE instruction. A 32-bit parameter is provided with the interruption to...
assist the program in determining the operation for which the interruption is reported.

Service signal is a floating interruption condition and is presented to the first CPU in the configuration which can perform the interruption. The interruption condition is cleared when it causes an interruption in any one of the CPUs and also by subsystem reset.

The subclass-mask bit is in bit position 54 of control register 0. This bit is initialized to zero.

The service-signal condition is indicated by an external-interruption code of 2401 hex. A 32-bit parameter is stored at real locations 128-131.

### I/O Interruption

The input/output (I/O) interruption provides a means by which the CPU responds to conditions originating in I/O devices and the channel subsystem.

A request for an I/O interruption may occur at any time, and more than one request may occur at the same time. The requests are preserved and remain pending until accepted by a CPU, or until cleared by some other means, such as subsystem reset.

The I/O interruption occurs at the completion of a unit of operation. Priority is established among requests so that in each CPU only one interruption request is processed at a time. Priority among requests for interruptions of differing I/O-interruption subclasses is according to the numerical value of the I/O-interruption subclass (with zero having the highest priority), in conjunction with the I/O-interruption subclass-mask settings in control register 6. For more details, see Chapter 16, “I/O Interruptions.”

When a CPU becomes enabled for I/O interruptions and the channel subsystem has established priority for a pending I/O-interruption condition, the interruption occurs at the completion of the instruction execution or interruption that causes the enabling.

An I/O interruption causes the old PSW to be stored at real locations 368-383 and a new PSW to be fetched from real locations 496-511. Additional information, in the form of an eight-byte I/O-interruption code, is stored at real locations 184-191. The I/O-interruption code consists of a 32-bit subsystem-identification word and a 32-bit interruption parameter.

An I/O interruption can occur only while a CPU is enabled for the interruption subclass presenting the request. The I/O-mask bit, bit 6 of the PSW, and the I/O-interruption subclass mask in control register 6 determine whether the CPU is enabled for a particular I/O interruption.

I/O interruptions are grouped into eight I/O-interruption subclasses, numbered from 0-7. Each I/O-interruption subclass has an associated I/O-interruption subclass-mask bit in bit positions 32-39 of control register 6. Each subchannel has an I/O-interruption subclass value associated with it. The CPU is enabled for I/O interruptions of a particular I/O-interruption subclass only when PSW bit 6 is one and the associated I/O-interruption subclass-mask bit in control register 6 is also one. If the corresponding I/O-interruption subclass-mask bit is zero, then the CPU is disabled for I/O interruptions with that subclass value. I/O interruptions for all subclasses are disallowed when PSW bit 6 is zero.

### Machine-Check Interruption

The machine-check interruption is a means for reporting to the program the occurrence of equipment malfunctions. Information is provided to assist the program in determining the source of the fault and extent of the damage.

A machine-check interruption causes the old PSW to be stored at real locations 352-367 and a new PSW to be fetched from real locations 480-495.

The cause and severity of the malfunction are identified by a 64-bit machine-check-interruption code stored at real locations 232-239. Further information identifying the cause of the interruption and the location of the fault may be stored at real locations 244-255 and 4608-5119.

The interruption action and the storing of the associated information are under the control of PSW bit 13 and bits in control register 14. See Chapter 11, “Machine-Check Handling” for more detailed information.
Program Interruption

Program interruptions are used to report exceptions and events which occur during execution of the program.

A program interruption causes the old PSW to be stored at real locations 336-351 and a new PSW to be fetched from real locations 464-479.

The cause of the interruption is identified by the interruption code. The interruption code is placed at real locations 142-143, the instruction-length code is placed in bit positions 5 and 6 of the byte at real location 141 with the rest of the bits set to zeros, and zeros are stored at real location 140. For some causes, additional information identifying the reason for the interruption is stored at real locations 144-183.

Except for PER events and the crypto-operation exception, the condition causing the interruption is indicated by a coded value placed in the rightmost seven bit positions of the interruption code. Only one condition at a time can be indicated. Bits 0-7 of the interruption code are set to zeros.

PER events are indicated by setting bit 8 of the interruption code to one. When this is the only condition, bits 0-7 and 9-15 are also set to zeros. When a PER event is indicated concurrently with another program-interruption condition, bit 8 is one, and bits 0-7 and 9-15 are set as for the other condition.

The crypto-operation exception is indicated by an interruption code of 0119 hex, or 0199 hex if a PER event is also indicated.

When there is a corresponding mask bit, a program interruption can occur only when that mask bit is one. The program mask in the PSW controls four of the exceptions, the IEEE masks in the FPC register control the IEEE exceptions, bit 33 in control register 0 controls whether SET SYSTEM MASK causes a special-operation exception, bits 48-63 in control register 8 control interruptions due to monitor events, and a hierarchy of masks control interruptions due to PER events. When any controlling mask bit is zero, the condition is ignored; the condition does not remain pending.

Programming Notes:

1. When the new PSW for a program interruption has a PSW-format error or causes an exception to be recognized in the process of instruction fetching, a string of program interruptions may occur. See "Priority of Interruptions" on page 6-48 for a description of how such strings are terminated.

2. Some of the conditions indicated as program exceptions may be recognized also by the channel subsystem, in which case the exception is indicated in the subchannel-status word or extended-status word.

Data-Exception Code (DXC)

When a data exception causes a program interruption, a data-exception code (DXC) is stored at location 147, and zeros are stored at locations 144-146. The DXC distinguishes between the various types of data-exception conditions. When the AFP-register (additional floating-point register) control bit, bit 45 of control register 0, is one, the DXC is also placed in the DXC field of the floating-point-control (FPC) register. The DXC field in the FPC register remains unchanged when any other program exception is reported. The DXC is an 8-bit code indicating the specific cause of a data exception. The data exceptions and data-exception codes are shown in Figure 6-2 on page 6-15 and Figure 6-3 on page 6-16.

Priority of Program Interruptions for Data Exceptions

When more than one data exception applies and is enabled, the exception with the smallest DXC value is reported. Thus, for example, DXC 2 (BFP instruction) takes precedence over any IEEE exception condition.

When both a specification exception and an AFP-register data exception apply, it is unpredictable which one is reported.
Program-Interruption Conditions

The following is a detailed description of each program-interruption condition.

Addressing Exception
An addressing exception is recognized when the CPU attempts to reference a main-storage location that is not available in the configuration. A main-storage location is not available in the configuration when the location is not installed, when the storage unit is not in the configuration, or when power is off in the storage unit. An address designating a storage location that is not available in the configuration is referred to as invalid.

The operation is suppressed when the address of the instruction is invalid. Similarly, the operation is suppressed when the address of the target instruction of EXECUTE is invalid. Also, the unit of operation is suppressed when an addressing exception is encountered in accessing a table or table entry. The tables and table entries to which the rule applies are the dispatchable-unit-control table, the primary ASN-second-table entry, and entries in the access list, region first table, region second table, region third table, segment table, page table, linkage table, entry table, ASN first table, ASN second table, authority table, linkage stack, and trace table. Addressing exceptions result in suppression when they are encountered for references to the region first table, region second table, region third table, segment table, and page table, in both implicit references for dynamic address translation and references associated with the execution of LOAD REAL ADDRESS, STORE REAL ADDRESS, and TEST PROTECTION. Similarly, addressing exceptions for accesses to the dispatchable-unit-control table, primary ASN-second-table entry, access list, ASN second table, or authority table result in suppression when they are encountered in access-register translation done either implicitly or as part of LOAD REAL ADDRESS, STORE REAL ADDRESS, TEST ACCESS, or TEST PROTECTION. Except for some specific instructions whose execution is suppressed, the operation is terminated for an operand address that can be translated but designates an unavailable location. See Figure 6-4 on page 6-17.

For termination, changes may occur only to result fields. In this context, the term “result field” includes the condition code, registers, and any storage locations that are provided and that are designated to be changed by the instruction. Therefore, if an instruction is due to change only the contents of a field in storage, and every byte of the field is in a location that is not available in the configuration, the operation is suppressed. When part of an operand location is available in the configuration and part is not, storing may be performed in the part that is available in the configuration.

When an addressing exception occurs during the fetching of an instruction or during the fetching of a DAT table entry associated with an instruction fetch, it is unpredictable whether the ILC is 1, 2, or 3. When the exception is associated with fetching the target of EXECUTE, the ILC is 2.

In all cases of addressing exceptions not associated with instruction fetching, the ILC is 1, 2, or 3, indicating the length of the instruction that caused the reference.

An addressing exception is indicated by a program-interruption code of 0005 hex (or 0085 hex if a concurrent PER event is indicated).
<table>
<thead>
<tr>
<th>Exception</th>
<th>Applicable Instruction Types</th>
<th>Effect of CR0.45</th>
<th>FPC Mask</th>
<th>FPC Flag</th>
<th>DXC (Binary)</th>
<th>Interruption Action</th>
<th>DXC Placed in Real Loc 147</th>
<th>DXC Placed in FPC Byte 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal operand</td>
<td>Decimal(^1)</td>
<td>0</td>
<td>none</td>
<td>none</td>
<td>00000000</td>
<td>Suppress or Terminate</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFP register</td>
<td>FPS &amp; HFP</td>
<td>0(^*)</td>
<td>none</td>
<td>none</td>
<td>00000001</td>
<td>Suppress</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BFP instruction</td>
<td>BFP</td>
<td>0(^*)</td>
<td>none</td>
<td>none</td>
<td>00000010</td>
<td>Suppress</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>IEEE invalid operation</td>
<td>BFP</td>
<td>1(^*)</td>
<td>0.0</td>
<td>1.0</td>
<td>10000000</td>
<td>Suppress</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IEEE division by zero</td>
<td>BFP</td>
<td>1(^*)</td>
<td>0.1</td>
<td>1.1</td>
<td>01000000</td>
<td>Suppress</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IEEE overflow</td>
<td>BFP</td>
<td>1(^*)</td>
<td>0.2</td>
<td>1.2</td>
<td>0010xy00</td>
<td>Complete</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IEEE underflow</td>
<td>BFP</td>
<td>1(^*)</td>
<td>0.3</td>
<td>1.3</td>
<td>0001xy00</td>
<td>Complete</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IEEE inexact</td>
<td>BFP</td>
<td>1(^*)</td>
<td>0.4</td>
<td>1.4</td>
<td>00001y00</td>
<td>Complete</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Explanation:**

\(^1\) Decimal-operand data exception applies to the decimal instructions (Chapter 8) and the general instructions COMPRESSION CALL and CONVERT TO BINARY (Chapter 7).

\(^0\) This exception is recognized only when CR0.45 is zero.

\(^1\) This exception is recognized only when CR0.45 is one.

\(^x\) For IEEE overflow and IEEE underflow, bits 4 and 5 of the DXC are set to 00, 10, or 11 binary, indicating that the result is exact, inexact and truncated, or inexact and incremented, respectively.

\(^y\) For IEEE inexact, bit 5 of the DXC is set to zero or one, indicating that the result is inexact and truncated or inexact and incremented, respectively.

**BFP** Binary-floating-point instructions (Chapter 19).

**FPS** Floating-point-support instructions (Chapter 9).

**HFP** Hexadecimal-floating-point instructions (Chapter 18).

*Figure 6-3. Data Exceptions*
Figure 6-4. Summary of Action for Addressing and Protection Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Action on</th>
<th>Table-Entry Fetch</th>
<th>Table-Entry Store</th>
<th>Instruction Fetch</th>
<th>Operand Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing exception</td>
<td></td>
<td>Suppress</td>
<td>Suppress</td>
<td>Suppress</td>
<td>Suppress for IPTE, LASP, LPSW, LPSWE, MSCH, PLO*, RP, SCKC, SPT, SPX, SSCH, SSM, STCRW, STNSM, STOSM, TPI, and TPROT Terminate for all others.⁴</td>
</tr>
<tr>
<td>Protection exception for key-controlled protection</td>
<td></td>
<td>--</td>
<td>--</td>
<td>Suppress</td>
<td>Suppress for IPTE, LASP, LPSW, LPSWE, MSCH, PLO*, RP, SCKC, SPT, SPX, SSCH, SSM, STCRW, STNSM, STOSM, and TPI⁵ Terminate for all others.⁴</td>
</tr>
<tr>
<td>Protection exception for access-list-controlled protection</td>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>Suppress</td>
</tr>
<tr>
<td>Protection exception for page protection</td>
<td></td>
<td>--</td>
<td>Suppress³</td>
<td>--</td>
<td>Suppress⁶</td>
</tr>
<tr>
<td>Protection exception for low-address protection</td>
<td></td>
<td>--</td>
<td>Suppress</td>
<td>--</td>
<td>Suppress for IPTE, STCRW, STNSM, STOSM, and TPI⁵ Terminate for all others.⁴</td>
</tr>
</tbody>
</table>

Explanation:

- Not applicable.

  ³ Table entries include region table, segment table, page table, linkage table, entry table, ASN first table, ASN second table, authority table, dispatchable-unit-control table, primary ASN-second-table entry, access list, and linkage stack.

- Table entries include linkage stack and trace table.

- Page protection applies to the linkage stack but not the trace table.

- For termination, changes may occur only to result fields. In this context, "result field" includes condition code, registers, and storage locations, if any, which are designated to be changed by the instruction. However, no change is made to a storage location or a storage key when the reference causes an access exception. Therefore, if an instruction is due to change only the contents of a field in main storage, and every byte of that field would cause an access exception, the result is the same as if the operation had been suppressed. The action may be, for key-controlled protection and low-address protection, suppression instead of termination; see "Suppression on Protection" in Chapter 3, "Storage."

- When the effective address of TPI is zero, the store access is to implicit real locations 184-191, and key-controlled protection, page protection, and low-address protection do not apply.

- Suppression occurs only for the compare-and-load and compare-and-swap operations.
AFX-Translation Exception
An AFX-translation exception is recognized when, during ASN translation in the space-switching form of PROGRAM RETURN, PROGRAM TRANSFER, or SET SECONDARY ASN, or during ASN translation in PROGRAM RETURN when the restored SASN does not equal the restored PASN, bit 0 of the ASN-first-table entry used is not zero.

The ASN being translated is stored at real locations 174 and 175, and real locations 172 and 173 are set with zeros.

The operation is nullified.

The instruction-length code is 1 or 2.

The AFX-translation exception is indicated by a program-interruption code of 0020 hex (or 00A0 hex if a concurrent PER event is indicated).

ALEN-Translation Exception
An ALEN-translation exception is recognized during access-register translation when either:

1. The access register used contains an access-list-entry number that designates an access-list entry which is beyond the end of the access list designated by the effective access-list designation.

2. Bit 0 of the access-list entry is not zero.

The number of the access register is stored in bit positions 4-7 of real location 160, and bits 0-3 of the location are set to zeros.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ALEN-translation exception is indicated by a program-interruption code of 0029 hex (or 00A9 hex if a concurrent PER event is indicated).

ALE-Sequence Exception
An ALE-sequence exception is recognized during access-register translation when the access register contains an access-list-entry sequence number (ALESN) which is not equal to the ALESN in the access-list entry that is designated by the access register.

The number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ALE-sequence exception is indicated by a program-interruption code of 002A hex (or 00AA hex if a concurrent PER event is indicated).

ALET-Specification Exception
An ALET-specification exception is recognized during access-register translation when bit positions 0-6 of the access-list-entry token in the access register used do not contain all zeros. However, when access-register 0 is used, except in TEST ACCESS, it is treated as containing all zeros, and this exception is not recognized. TEST ACCESS uses the actual contents of access register 0.

The operation is suppressed.

The instruction-length code is 1, 2, or 3.

The ALET-specification exception is indicated by a program-interruption code of 0028 hex (or 00A8 hex if a concurrent PER event is indicated).

ASCE-Type Exception
An ASCE-type exception is recognized when any of the following is true during dynamic address translation:

1. The address-space-control element being used is a region-second-table designation, and bits 0-10 of the virtual address being translated are not all zeros.

2. The address-space-control element being used is a region-third-table designation, and bits 0-21 of the virtual address being translated are not all zeros.

3. The address-space-control element being used is a segment-table designation, and bits 0-32 of the virtual address being translated are not all zeros.

The exception is recognized as part of the execution of the instruction that needs the translation of an instruction or operand address, except for the operand address in LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.
When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-58 for a detailed description of this information.

The unit of operation is nullified.

The instruction-length code is 1, 2, or 3.

The ASCE-type exception is indicated by a program-interruption code of 0038 hex (or 00B8 hex if a concurrent PER event is indicated).

ASTE-Instance Exception

An ASTE-instance exception is recognized when the ASN-and-LX-reuse facility is installed, and any of the following is true:

1. During ASN translation in PROGRAM TRANSFER WITH INSTANCE with space switching, the ASTEIN in bit positions 0-31 of general register R1 is not equal to the ASTEIN in the located ASTE.
2. During PASN translation in PROGRAM RETURN with space switching, the ASN-and-LX-reuse control in control register 0 is one and the PASTEIN in the linkage-stack state entry is not equal to the ASTEIN in the located ASTE.
3. During ASN translation in SET SECONDARY ASN WITH INSTANCE with space switching, the ASTEIN in bit positions 0-31 of general register R1 is not equal to the ASTEIN in the located ASTE.
4. During SASN translation in PROGRAM RETURN to current primary or with space switching, the ASN-and-LX-reuse control in control register 0 is one and the SASTEIN in the linkage-stack state entry is not equal to the ASTEIN in the located ASTE.

Information is stored as follows:

- In cases 1 and 2, bit 2 of real location 160 is set to one, and bits 0, 1, and 3-7 are set to zeros.
- In cases 3 and 4, bit 3 of real location 160 is set to one, and bits 0-2 and 4-7 are set to zeros.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ASTE-instance exception is indicated by a program-interruption code of 002F hex (or 00AF hex if a concurrent PER event is indicated).

ASTE-Sequence Exception

An ASTE-sequence exception is recognized when any of the following is true:

1. During access-register translation, except as in 2, the access-list entry used contains an ASN-second-table-entry sequence number (ASTESN) which is not equal to the ASTESN in the ASN-second-table entry that is designated by the access-list entry. The access-list entry is the one designated by the access register used.
2. During access-register translation of ALET 1 by BRANCH IN SUBSPACE GROUP, the subspace ASTESN (SSASTESN) in the dispatchable-unit control table (DUCT) is not equal to the ASTESN in the subspace ASTE designated by the subspace-ASTE origin (SSASTEO) in the DUCT.
3. During a subspace-replacement operation, the subspace ASTESN (SSASTESN) in the dispatchable-unit control table (DUCT) is not equal to the ASTESN in the subspace ASTE designated by the subspace-ASTE origin (SSASTEO) in the DUCT.

In the first and second cases, the number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros. In the third case, all zeros are stored at real location 160.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ASTE-sequence exception is indicated by a program-interruption code of 002C hex (or 00AC hex if a concurrent PER event is indicated).

Programming Note: The storing of zeros at real location 160 in the case of an ASTE-sequence exception recognized during a subspace-replacement operation is a unique indication since the use of access register 0 in access-register translation cannot result in the exception.
ASTE-Validity Exception
An ASTE-validity exception is recognized when any of the following is true:

1. During access-register translation, except as in 2, the access-list entry used designates an ASN-second-table entry in which bit 0 is not zero. The access-list entry is the one designated by the access register used.

2. During access-register translation of ALET 1 by BRANCH IN SUBSPACE GROUP, the subspace-ASTE origin (SSASTEO) in the dispatchable-unit control table designates an ASN-second-table entry in which bit 0 is not zero.

3. During a subspace-replacement operation, the subspace-ASTE origin (SSASTEO) in the dispatchable-unit control table designates an ASN-second-table entry in which bit 0 is not zero.

In the first and second cases, the number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros. In the third case, all zeros are stored at real location 160.

The operation is nullified.

The instruction-length code is 1, 2, or 3.

The ASTE-validity exception is indicated by a program-interruption code of 002B hex (or 00AB hex if a concurrent PER event is indicated).

Programming Note: The storing of zeros at real location 160 in the case of an ASTE-validity exception recognized during a subspace-replacement operation is a unique indication since the use of access register 0 in access-register translation cannot result in the exception.

ASX-Translation Exception
An ASX-translation exception is recognized when, during execution of the space-switching form of PROGRAM CALL, during ASN translation in the space-switching form of PROGRAM RETURN, PROGRAM TRANSFER, or SET SECONDARY ASN, or during ASN translation in PROGRAM RETURN when the restored SASN does not equal the restored PASN, bit 0 of the ASN-second-table entry used is not zero.

The ASN being translated is stored at real locations 174 and 175, and real locations 172 and 173 are set with zeros.

The operation is nullified.

The instruction-length code is 1 or 2.

The ASX-translation exception is indicated by a program-interruption code of 0021 hex (or 00A1 hex if a concurrent PER event is indicated).

Crypto-Operation Exception
A crypto-operation exception is recognized when a crypto-facility instruction is executed while bit 61 of control register 0 is zero on a CPU which has the crypto facility installed and available. The crypto-operation exception is also recognized when a crypto-facility instruction is executed and the crypto facility is not installed or available on this CPU, but the facility can be made available to the program either on this CPU or another CPU in the configuration.

When a crypto-facility instruction is executed and the crypto facility is not installed on any CPU which is or can be placed in the configuration, it depends on the model whether a crypto-operation exception or an operation exception is recognized.

When a crypto-facility instruction is executed and the crypto facility is not installed on any CPU which is or can be placed in the configuration, it depends on the model whether a crypto-operation exception or an operation exception is recognized.

The operation is nullified when the crypto-operation exception is recognized.

The instruction-length code is 2.

The crypto-operation exception is indicated by a program-interruption code of 0119 hex (or 0199 hex if a concurrent PER event is indicated).

Data Exception
The data-exception conditions are shown in Figure 6-3 on page 6-16. A mask bit may or may not control whether an interruption occurs, as noted for each condition.

When a non-maskable data-exception condition is recognized, a program interruption for a data exception always occurs.

Each of the IEEE exception conditions is controlled by a mask bit in the floating-point-control (FPC) register. The handling of these conditions is described in the section IEEE Exception Conditions on page 19-10.
A data exception is recognized for the following cases:

- **Decimal-operand** data exception is recognized when an instruction which operates on decimal operands encounters invalid decimal digit or sign codes or has its operands specified improperly. The operation is suppressed, except that, for EDIT and EDIT AND MARK, the operation is terminated. See the section “Decimal-Operand Data Exception” on page 8-4 for details. A decimal-operand data exception is also recognized when COMPRESSION CALL encounters errors in its dictionaries, and, in this case, the operation is terminated. The decimal-operand data exception is reported with DXC 0.

- **AFP-register** data exception is recognized when bit 45 of control register 0 is zero, and a floating-point-support (FPS) instruction or a hexadecimal-floating-point (HFP) instruction specifies a floating-point register other than 0, 2, 4, or 6. The operation is suppressed and is reported with DXC 1.

- **BFP-instruction** data exception is recognized when bit 45 of control register 0 is zero and a BFP instruction is executed. The operation is suppressed and is reported with DXC 2.

- **IEEE-exception-condition** data exceptions are recognized when a BFP instruction encounters an exceptional condition. The operation is suppressed or completed, depending on the type of condition. See the section “IEEE Exception Conditions” on page 19-10 for details.

The instruction-length code is 1, 2, or 3.

The data exception is indicated by a program-interruption code of 0007 hex (or 0087 hex if a concurrent PER event is indicated).

**Decimal-Divide Exception**
A decimal-divide exception is recognized when in decimal division the divisor is zero or the quotient exceeds the specified data-field size.

The decimal-divide exception is indicated only if the sign codes of both the divisor and dividend are valid and only if the digit or digits used in establishing the exception are valid.

The operation is suppressed.

The instruction-length code is 2 or 3.

The decimal-divide exception is indicated by a program-interruption code of 000B hex (or 008B hex if a concurrent PER event is indicated).

**Decimal-Overflow Exception**
A decimal-overflow exception is recognized when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the result.

The interruption may be disallowed by the decimal-overflow mask (PSW bit 21).

The operation is completed. The result is obtained by ignoring the overflow digits, and condition code 3 is set.

The instruction-length code is 2 or 3.

The decimal-overflow exception is indicated by a program-interruption code of 000A hex (or 008A hex if a concurrent PER event is indicated).

**Execute Exception**
The execute exception is recognized when the target instruction of EXECUTE is another EXECUTE.

The operation is suppressed.

The instruction-length code is 2.

The execute exception is indicated by a program-interruption code of 0003 hex (or 0083 hex if a concurrent PER event is indicated).

**EX-Translation Exception**
An EX-translation exception is recognized during PC-number translation in PROGRAM CALL when the entry-table entry designated by the entry-index part of the PC number is beyond the end of the entry table as designated by the linkage-table or linkage-second-table entry used.

When ASN-and-LX reuse is not installed or is not enabled by a one value of the ASN-and-LX-reuse control in control register 0, or if it is installed and enabled and bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the address (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at the instruction.
real location 172. When ASN-and-LX reuse is installed and enabled and bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2.

The EX-translation exception is indicated by a program-interruption code of 0023 hex (or 00A3 hex if a concurrent PER event is indicated).

Extended-Authority Exception
An extended-authority exception is recognized during access-register translation when all of the following are true:

1. The private bit in the access-list entry used is one.
2. The access-list-entry authorization index (ALEAX) in the access-list entry is not equal to the extended authorization index (EAX) in control register 8.
3. Either of the following is true:
   a. The authority-table entry designated by the EAX is beyond the length of the authority table used. The authority table is the one designated by the ASN-second-table entry that is designated by the access-list entry used.
   b. The secondary-authority bit designated by the EAX is zero.

The access-list entry is the one designated by the access register used.

The number of the access register is stored in bit positions 4-7 at real location 160, and bits 0-3 are set to zeros.

The extended-authority exception is indicated by a program-interruption code of 002D hex (or 00AD hex if a concurrent PER event is indicated).

Fixed-Point-Divide Exception
A fixed-point-divide exception is recognized when any of the following is true:

1. In signed or unsigned binary division when the result is defined to be 32 bits, the divisor is zero, or the quotient cannot be expressed as a 32-bit signed or unsigned, respectively, binary integer.
2. In signed or unsigned binary division when the result is defined to be 64 bits, the divisor is zero, or the quotient cannot be expressed as a 64-bit signed or unsigned, respectively, binary integer.
3. The result of CONVERT TO BINARY cannot be expressed as a 32-bit signed binary integer for a 32-bit result or as a 64-bit signed binary integer for a 64-bit result.

In the case of division, the operation is suppressed. The execution of CONVERT TO BINARY (CVB) is completed by ignoring the leftmost bits that cannot be placed in the register. The execution of CONVERT TO BINARY (CVBG) is suppressed.

The instruction-length code is 1, 2, or 3.

The fixed-point-divide exception is indicated by a program-interruption code of 0009 hex (or 0089 hex if a concurrent PER event is indicated).

Fixed-Point-Overflow Exception
A fixed-point-overflow exception is recognized when an overflow occurs during signed binary arithmetic or signed left-shift operations.

The interruption may be disallowed by the fixed-point-overflow mask (PSW bit 20).

The operation is completed. The result is obtained by ignoring the overflow information, and condition code 3 is set.

The instruction-length code is 1, 2, or 3.

The fixed-point-overflow exception is indicated by a program-interruption code of 0008 hex (or 0088 hex if a concurrent PER event is indicated).
HFP-Divide Exception
An HFP-divide exception is recognized when in HFP division the divisor has a zero fraction.

The operation is suppressed.

The instruction-length code is 1 or 2.

The HFP-divide exception is indicated by a program-interruption code of 000F hex (or 008F hex if a concurrent PER event is indicated).

HFP-Exponent-Overflow Exception
An HFP-exponent-overflow exception is recognized when the result characteristic of an HFP operation exceeds 127 and the result fraction is not zero.

The operation is completed. The fraction is normalized, and the sign and fraction of the result remain correct. The result characteristic is made 128 smaller than the correct characteristic.

The instruction-length code is 1, 2, or 3.

The HFP-exponent-overflow exception is indicated by a program-interruption code of 000C hex (or 008C hex if a concurrent PER event is indicated).

HFP-Exponent-Underflow Exception
An HFP-exponent-underflow exception is recognized when the result characteristic of an HFP operation is less than zero and the result fraction is not zero. For an extended-format HFP result, HFP-exponent underflow is indicated only when the high-order characteristic underflows.

The interruption may be disallowed by the HFP-exponent-underflow mask (PSW bit 22).

The operation is completed. The HFP-exponent-underflow mask also affects the result of the operation. When the mask bit is zero, the sign, characteristic, and fraction are set to zero, making the result a true zero. When the mask bit is one, the fraction is normalized, the characteristic is made 128 larger than the correct characteristic, and the sign and fraction remain correct.

The instruction-length code is 1, 2, or 3.

The HFP-exponent-underflow exception is indicated by a program-interruption code of 000D hex (or 008D hex if a concurrent PER event is indicated).

HFP-Significance Exception
An HFP-significance exception is recognized when the result fraction in HFP addition or subtraction is zero.

The interruption may be disallowed by the HFP-significance mask (PSW bit 23).

The operation is completed. The HFP-significance mask also affects the result of the operation. When the mask bit is zero, the operation is completed by replacing the result with a true zero. When the mask bit is one, the operation is completed without further change to the characteristic of the result.

The instruction-length code is 1 or 2.

The HFP-significance exception is indicated by a program-interruption code of 000E hex (or 008E hex if a concurrent PER event is indicated).

HFP-Square-Root Exception
An HFP-square-root exception is recognized when the second operand of an HFP SQUARE ROOT instruction is less than zero.

The operation is suppressed.

The instruction-length code is 2 or 3.

The HFP-square-root exception is indicated by a program-interruption code of 001D hex (or 009D hex if a concurrent PER event is indicated).

LFX-Translation Exception
An LFX-translation exception may be recognized only when the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse control in control register 0 is one.

An LFX-translation exception is recognized during PC-number translation in PROGRAM CALL when either:

1. The linkage-first-table entry specified by the linkage-first-index part of the PC number is beyond the end of the linkage first table as designated by the linkage-first-table designation used.

2. Bit 0 of the linkage-first-table entry is not zero.
When bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the address (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172. When bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2.

The LFX-translation exception is indicated by a program-interruption code of 0026 hex (or 00A6 hex if a concurrent PER event is indicated).

LSTE-Sequence Exception
An LSTE-sequence exception may be recognized only when the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse control in control register 0 is one.

An LSTE-sequence exception is recognized during PC-number translation in PROGRAM CALL when the linkage-second-table-entry sequence number (LSTESN) in the linkage-second-table entry used is nonzero and not equal to the LSTESN in bit positions 0-31 of general register 15.

When bit 44 of the second-operand address used by PROGRAM CALL is zero, bits 44-63 of the address (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172. When bit 44 of the second-operand address is one, bits 32-63 of the address (a 32-bit PC number) are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2.

The LSTE-sequence exception is indicated by a program-interruption code of 002E hex (or 00AE hex if a concurrent PER event is indicated).

LSX-Translation Exception
An LSX-translation exception may be recognized only when the ASN-and-LX-reuse facility is not installed or the ASN-and-LX-reuse control in control register 0 is zero.

An LSX-translation exception is recognized during PC-number translation in PROGRAM CALL when either:

1. The linkage-table entry designated by the linkage-index part of the PC number is beyond the end of the linkage table as designated by the linkage-table designation used.

2. Bit 0 of the linkage-table entry is not zero.

Bits 44-63 of the second-operand address used by PROGRAM CALL (a 20-bit PC number), with 12 zeros appended on the left, are stored in the word at real location 172.

The operation is nullified.

The instruction-length code is 2.

The LSX-translation exception is indicated by a program-interruption code of 0027 hex (or 00A7 hex if a concurrent PER event is indicated).

LX-Translation Exception
An LX-translation exception may be recognized only when the ASN-and-LX-reuse facility is not installed or the ASN-and-LX-reuse control in control register 0 is zero.

An LX-translation exception is recognized during PC-number translation in PROGRAM CALL when either:

1. The linkage-table entry designated by the linkage-index part of the PC number is beyond the end of the linkage table as designated by the linkage-table designation used.

2. Bit 0 of the linkage-table entry is not zero.

The operation is nullified.

The instruction-length code is 2.

The LX-translation exception is indicated by a program-interruption code of 0022 hex (or 00A2 hex if a concurrent PER event is indicated).
Monitor Event

A monitor event is recognized when MONITOR CALL is executed and the monitor-mask bit in control register 8 corresponding to the class specified by instruction bits 12-15 is one. The information in control register 8 has the following format:

Control Register 8

<table>
<thead>
<tr>
<th>Monitor Masks</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 63</td>
</tr>
</tbody>
</table>

The monitor-mask bits, bits 48-63 of control register 8, correspond to monitor classes 0-15, respectively. Any number of monitor-mask bits may be on at a time; together they specify the classes of monitor events that are monitored at that time. The mask bits are initialized to zeros.

When MONITOR CALL is executed and the corresponding monitor-mask bit is one, a program interruption for monitor event occurs.

Additional information is stored at real locations 148-149 and 176-183. The format of the information stored at these locations is as follows:

Real Locations 148-149

<table>
<thead>
<tr>
<th>Monitor Class No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>0 8 15</td>
</tr>
</tbody>
</table>

Real Locations 176-183

<table>
<thead>
<tr>
<th>Monitor Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

The contents of bit positions 8-15 of the MONITOR CALL instruction are stored at real location 149 and constitute the monitor-class number. Zeros are stored at real location 148. The effective address specified by the B₁ and D₁ fields of the instruction forms the monitor code, which is stored in the doubleword at real location 176. The value of the address is under control of the addressing mode, bits 31 and 32 of the current PSW. In the 24-bit addressing mode, bits 0-39 of the address are zeros, while in the 31-bit addressing mode, bits 0-32 are zeros.

The operation is completed.

Operand Exception

An operand exception is recognized when any of the following is true:

1. Execution of CLEAR SUBCHANNEL, HALT SUBCHANNEL, MODIFY SUBCHANNEL, RESUME SUBCHANNEL, START SUBCHANNEL, STORE SUBCHANNEL, or TEST SUBCHANNEL is attempted and bit positions 32-47 of general register 1 do not contain 0001 hex. However, an exception due to ones in bit positions 32-39 of the register may or may not be recognized.

2. Execution of MODIFY SUBCHANNEL is attempted, and bits 1 and 6 of word 1 of the SCHIB operand are not zeros or bits 9-10 and 25-30 of word 6 of the SCHIB operand are not all zeros.

3. Execution of MODIFY SUBCHANNEL is attempted, and bits 9 and 10 of word 1 of the SCHIB operand are both one.

4. Execution of RESET CHANNEL PATH is attempted, and bits 40-55 of general register 1 are not all zeros.

5. Execution of SET ADDRESS LIMIT is attempted, and bits 32 and 48-63 of general register 1 are not all zeros.

6. Execution of SET CHANNEL MONITOR is attempted, bit 62 of general register 1 is one, and bits 59-63 of general register 2 are not all zeros.

7. Execution of SET CHANNEL MONITOR is attempted, and bits 36-61 of general register 1 are not all zeros.

8. Execution of START SUBCHANNEL is attempted, and bits 5, 13, and 25-28 of word 1 of the ORB operand are not all zeros.

9. Execution of START SUBCHANNEL is attempted, and bit 11 of word 1 of the ORB operand is not zero. This exception may or may not be recognized.

The operation is suppressed.
The instruction-length code is 2.

The operand exception is indicated by a program-interruption code of 0015 hex (or 0095 hex if a concurrent PER event is indicated).

**Operation Exception**
An operation exception is recognized when the CPU attempts to execute an instruction with an invalid operation code. The operation code may be unassigned, or the instruction with that operation code may not be installed on the CPU.

The operation is suppressed.

The instruction-length code is 1, 2, or 3.

The operand exception is indicated by a program-interruption code of 0001 hex (or 0081 hex if a concurrent PER event is indicated).

**Programming Notes:**
1. Some models may offer instructions not described in this publication, such as those provided for assists or as part of special or custom features. Consequently, operation codes not described in this publication do not necessarily cause an operation exception to be recognized. Furthermore, these instructions may cause modes of operation to be set up or may otherwise alter the machine so as to affect the execution of subsequent instructions. To avoid causing such an operation, an instruction with an operation code not described in this publication should be executed only when the specific function associated with the operation code is desired.

2. The operation code 00, with a two-byte instruction format, currently is not assigned. It is improbable that this operation code will ever be assigned.

**Page-Translation Exception**
A page-translation exception is recognized when the page-invalid bit is one.

The exception is recognized as part of the execution of an instruction that needs the page-table entry in the translation of an instruction or operand address, except for the operand address in LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code, and except for an operand address in MOVE PAGE, in which case the condition is indicated by the setting of the condition code if the condition-code-option bit, bit 55 of general register 0, is one.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See "Assigned Storage Locations" on page 3-58 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during a reference to the target of EXECUTE, the ILC is 2.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The page-translation exception is indicated by a program-interruption code of 0011 hex (or 0091 hex if a concurrent PER event is indicated).

**PC-Translation-Specification Exception**
A PC-translation-specification exception is recognized during PC-number translation in PROGRAM CALL when either of the following is true for the entry-table entry (ETE) used:

1. The PROGRAM CALL operation is the basic operation (bit 128 of the ETE is zero) in the 24-bit or 31-bit addressing mode (bit 31 of the PSW is zero), bit 32 of the ETE is zero (specifying the 24-bit mode), and bits 33-39 of the ETE are not all zeros.

2. The PROGRAM CALL operation is the stacking operation (bit 128 of the ETE is one), bits 32 and 129 of the ETE are zeros (specifying the 24-bit mode), and bits 33-39 of the ETE are not all zeros.

The operation is suppressed.

The instruction-length code is 2.

The PC-translation-specification exception is indicated by a program-interruption code of 001F hex (or 009F hex if a concurrent PER event is indicated).
PER Event
A PER event is recognized when the CPU is enabled for PER and one or more of these events occur.

The PER mask, bit 1 of the PSW, controls whether the CPU is enabled for PER. When the PER mask is zero, PER events are not recognized. When the bit is one, PER events are recognized, subject to the PER-event-mask bits in control register 9.

The unit of operation is completed, unless another condition has caused the unit of operation to be inhibited, nullified, suppressed, or terminated.

Information identifying the event is stored at real locations 150-159 and conditionally at real location 161.

The instruction-length code is 0, 1, 2, or 3. Code 0 is set only if a specification exception is indicated concurrently.

The PER event is indicated by setting bit 8 of the program-interruption code to one.

See "Program-Event Recording" on page 4-27 for a detailed description of the PER event and the associated interruption information.

Primary-Authority Exception
A primary-authority exception is recognized during ASN authorization in PROGRAM TRANSFER with space switching (PT-ss) when either:

1. The authority-table entry indicated by the authorization index in control register 4 is beyond the end of the authority table used. The authority table is the one designated by the ASN-second-table entry for the ASN used.
2. The primary-authority bit indicated by the authorization index is zero.

The ASN used is stored at real locations 174-175, and real locations 172-173 are set to zeros.

The operation is nullified.

The instruction-length code is 2.

The primary-authority exception is indicated by a program-interruption code of 0024 hex (or 00A4 hex if a concurrent PER event is indicated).

Privileged-Operation Exception
A privileged-operation exception is recognized when any of the following is true:

1. Execution of a privileged instruction is attempted in the problem state.
2. The value of the rightmost bit of the general register designated by the R2 field of the PROGRAM TRANSFER instruction is zero and would cause the PSW problem-state bit to change from the problem state (one) to the supervisor state (zero).
3. In the problem state, the key value specified by the second operand of the SET PSW KEY FROM ADDRESS instruction corresponds to a zero PSW-key-mask bit in control register 3.
4. In the problem state, the key value specified by the rightmost byte of the register designated by the R3 field of the MOVE WITH KEY instruction corresponds to a zero PSW-key-mask bit in control register 3.
5. In the problem state, the key value specified by the rightmost byte of the register designated by the R3 field for the instruction MOVE TO PRIMARY, MOVE TO SECONDARY, or MOVE WITH KEY corresponds to a zero PSW-key-mask bit in control register 3.
6. In the problem state, any of the instructions
   - EXTRACT PRIMARY ASN
   - EXTRACT SECONDARY ASN
   - INSERT ADDRESS SPACE CONTROL
   - INSERT PSW KEY
   - INSERT VIRTUAL STORAGE KEY
   is encountered, and the extraction-authority control, bit 4 of control register 0, is zero.
7. In the problem state, the result of ANDing the authorization key mask (AKM) with the PSW-key mask in control register 3 during PROGRAM CALL produces a result of zero.
8. In the problem state, bits 20-23 of the second-operand address of the SET ADDRESS SPACE CONTROL or SET ADDRESS SPACE CONTROL FAST instruction have the value 0011 binary.
9. In the problem state, the key value specified by the rightmost byte of general register 1 for the instruction MOVE WITH SOURCE KEY or MOVE WITH DESTINATION KEY corre-
sponds to a zero PSW-key-mask bit in control register 3.

10. In the problem state, the key value specified by the rightmost byte of the register designated by the R1 field for the instruction BRANCH AND SET AUTHORITY corresponds to a zero PSW-key-mask bit in control register 3.

11. In the problem state, bits 16 and 17 of the PSW field in the second operand of RESUME PROGRAM have the value 11 binary.

The operation is suppressed.

The instruction-length code is 2 or 3.

The privileged-operation exception is indicated by a program-interruption code of 0002 hex (or 0082 hex if a concurrent PER event is indicated).

**Protection Exception**

A protection exception is recognized when any of the following is true:

1. **Key-Controlled Protection**: The CPU attempts to access a storage location that is protected against the type of reference, and the access key does not match the storage key.

2. **Access-List-Controlled Protection**: The CPU attempts to store, in the access-register mode, by means of an access-list entry which has the fetch-only bit set to one.

3. **Low-Address Protection**: The CPU attempts a store that is subject to low-address protection, the effective address is in the range 0-511 or 4096-4607, and the low-address protection control, bit 35 of control register 0, is one.

4. **Page Protection**: The CPU attempts to store, with DAT on, into a page which has the page-protection bit set to one in either the page-table entry or the segment-table entry used in the translation.

The operation is suppressed when the location of the instruction is protected against fetching. Similarly, the operation is suppressed when the location of the target instruction of EXECUTE is protected against fetching.

For access-list-controlled protection and page-protection, the operation is suppressed. For the other two types of protection, except in the case of some specific instructions whose execution is suppressed, the operation is terminated when a protection exception is encountered during a reference to an operand location. See Figure 6-4 on page 6-17. However, the operation may be suppressed as described in “Suppression on Protection” on page 3-12.

For termination, changes may occur only to result fields. In this context, the term “result field” includes condition code, registers, and storage locations, if any, which are due to be changed by the instruction. However, no change is made to a storage location when a reference to that location causes a protection exception. Therefore, if an instruction is due to change only the contents of a field in storage, and every byte of that field would cause a protection exception, the operation is suppressed. When termination occurs on fetching, the protected information is not loaded into an addressable register nor moved to another storage location.

Information about the address causing the exception is stored at real locations 168-175 and conditionally at real location 160. See "Suppression on Protection" on page 3-12.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2.

For a protected operand location, the instruction-length code (ILC) is 1, 2, or 3, indicating the length of the instruction that caused the reference.

The protection exception is indicated by a program-interruption code of 0004 hex (or 0084 hex if a concurrent PER event is indicated).

**Region-First-Translation Exception**

A region-first-translation exception is recognized when a region first table is in the translation path for translation of a virtual address and either:

1. The region-first-table entry indicated by the region-first-index portion of the virtual address is outside the region first table.

2. The region-invalid bit is one.

The exception is sometimes called simply a region-translation exception, which term applies also to a region-second-translation exception and a region-third-translation exception.
The exception is recognized as part of the execution of an instruction that needs the region-first-table entry in the translation of an instruction or operand address, except for the operand address in LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-58 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The region-first-translation exception is indicated by a program-interruption code of 0039 hex (or 00B9 hex if a concurrent PER event is indicated).

Region-Second-Translation Exception
A region-second-translation exception is recognized when a region second table is in the translation path for translation of a virtual address and either:

1. The region-second-table entry indicated by the region-second-index portion of the virtual address is outside the region second table.

2. The region-invalid bit is one.

The exception is sometimes called simply a region-translation exception, which term applies also to a region-first-translation exception and a region-second-translation exception.

The exception is recognized as part of the execution of an instruction that needs the region-second-table entry in the translation of an instruction or operand address, except for the operand address in LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-58 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The region-second-translation exception is indicated by a program-interruption code of 003A hex (or 00BA hex if a concurrent PER event is indicated).

Region-Third-Translation Exception
A region-third-translation exception is recognized when a region third table is in the translation path for translation of a virtual address and either:

1. The region-third-table entry indicated by the region-third-index portion of the virtual address is outside the region third table.

2. The region-invalid bit is one.

The exception is sometimes called simply a region-translation exception, which term applies also to a region-first-translation exception and a region-second-translation exception.

The exception is recognized as part of the execution of an instruction that needs the region-third-table entry in the translation of an instruction or operand address, except for the operand address in LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See “Assigned Storage Locations” on page 3-58 for a detailed description of this information.
The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The region-third-translation exception is indicated by a program-interruption code of 003B hex (or 00BB hex if a concurrent PER event is indicated).

**Secondary-Authority Exception**

A secondary-authority exception is recognized during ASN authorization in SET SECONDARY ASN with space switching, or during ASN authorization in PROGRAM RETURN when the restored SASN does not equal the restored PASN, when either:

1. The authority-table entry indicated by the authorization index in control register 4 is beyond the end of the authority table used. The authority table is the one designated by the ASN-second-table entry for the ASN used. For PROGRAM RETURN, the ASN is the SASN being restored from the linkage-stack state entry used.
2. The secondary-authority bit indicated by the authorization index is zero.

The ASN used is stored at real locations 174-175, and real locations 172-173 are set to zeros.

The operation is nullified.

The instruction-length code is 1 or 2.

The secondary-authority exception is indicated by a program-interruption code of 0025 hex (or 00A5 hex if a concurrent PER event is indicated).

**Segment-Translation Exception**

A segment-translation exception is recognized when either:

1. The segment-table entry indicated by the segment-index portion of a virtual address is outside the segment table.
2. The segment-invalid bit is one.

The exception is recognized as part of the execution of an instruction that needs the segment-table entry in the translation of an instruction or operand address, except for the operand address in LOAD REAL ADDRESS and TEST PROTECTION, in which case the condition is indicated by the setting of the condition code.

When an interruption occurs, information about the virtual address causing the exception is stored at real locations 168-175 and conditionally at real locations 160 and 162. See Assigned Storage Locations on page 3-58 for a detailed description of this information.

The unit of operation is nullified.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The segment-translation exception is indicated by a program-interruption code of 0010 hex (or 0090 hex if a concurrent PER event is indicated).

**Space-Switch Event**

A space-switch event is recognized at the completion of the operation in each of the following cases:

1. The space-switching form of PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, or PROGRAM TRANSFER WITH INSTANCE is executed and any of the following is true:
   a. The primary space-switch-event-control bit, bit 57 of control register 1, is one before the operation.
   b. The primary space-switch-event-control bit is one after the operation.
   c. A PER event is indicated.

2. RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST is executed, the CPU is in the home-space mode either before or after the operation, but not both before and
after the operation, and any of the following is true:

a. The primary space-switch-event-control bit, bit 57 of control register 1, is one.

b. The home space-switch-event-control bit, bit 57 of control register 13, is one.

c. A PER event is indicated.

For PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, and PROGRAM TRANSFER WITH INSTANCE, and for a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the translation mode to the home-space mode, the old PASN, which is in bit positions 48-63 of control register 4 before the operation, is stored at real locations 174-175, and the old primary space-switch-event-control bit is placed in bit position 0 and zeros are placed in bit positions 1-15 at real locations 172-173.

For a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes the translation mode away from the home-space mode, zeros are stored at real locations 174-175, and the home space-switch-event-control bit is placed in bit position 0 and zeros are placed in bit positions 1-15 at real locations 172-173.

For a PROGRAM RETURN instruction that introduces a PSW-format error, it is unpredictable whether the instruction-length code is 0 or 1, or 0 or 2 if EXECUTE was used.

The operation is completed.

The instruction-length code is 0, 1, or 2.

The space-switch event is indicated by a program-interruption code of 001C hex (or 009C hex if a concurrent PER event is indicated).

Programming Notes:

1. The space-switch event permits the control program to gain control whenever a program enters or leaves a particular address space. The primary space-switch-event-control bit is loaded into control register 1, along with the remaining bits of the primary address-space-control element, whenever control register 1 is loaded.

2. The space-switch event may be useful in obtaining programmed authorization checking, in causing additional trace information to be recorded, or in enabling or disabling the CPU for PER or tracing.

3. Bit 121 of the ASN-second-table entry (ASTE) is loaded into bit position 57 of control register 1 as part of the PC-ss, PR-ss, PT-ss, and PTI-ss operations. If bit 121 of the ASTE for a particular address space is set to one, then a space-switch event is recognized when a program enters or leaves the address space by means of any of PC-ss, PR-ss, PT-ss, or PTI-ss.

4. The occurrence of a space-switch event at the completion of a PC-ss, PR-ss, PT-ss, or PTI-ss operation when any PER event is indicated, or at the completion of execution of a RESUME PROGRAM, SET ADDRESS SPACE CONTROL, or SET ADDRESS SPACE CONTROL FAST instruction that changes to or from the home-space mode when any PER event is indicated, permits the control program to determine the address space from which the instruction causing the PER event was fetched.

Special-Operation Exception

A special-operation exception is recognized when any of the following is true:

1. Execution of SET SYSTEM MASK is attempted in the supervisor state, and the SSM-suppression control, bit 33 of control register 0, is one.

2. Execution of any of the following instructions is attempted with DAT off:
   - EXTRACT PRIMARY ASN
   - EXTRACT SECONDARY ASN
   - INSERT ADDRESS SPACE CONTROL
   - INSERT VIRTUAL STORAGE KEY
   - SET ADDRESS SPACE CONTROL
   - SET SECONDARY ASN

3. Execution of MOVE TO PRIMARY or MOVE TO SECONDARY is attempted, and the CPU is not in the primary-space or secondary-space mode.

4. Execution of basic PROGRAM CALL or PROGRAM TRANSFER is attempted, and the CPU is not in the primary-space mode.
5. Execution of BRANCH AND STACK, stacking PROGRAM CALL, PROGRAM RETURN, or TRAP is attempted, and the CPU is not in the primary-space or access-register mode.

6. Execution of EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE or MODIFY STACKED STATE is attempted, and the CPU is not in the primary-space, access-register, or home-space mode.

7. Execution of LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL with space switching (PC-ss), PROGRAM TRANSFER with space switching (PT-ss), or SET SECONDARY ASN (SSAR-cp or SSAR-ss) is attempted, or execution of a PROGRAM RETURN instruction requiring PASN or SASN translation is attempted, and the ASN-translation control, bit 34 of control register 14, is zero.

8. Execution of PROGRAM CALL or PROGRAM TRANSFER is attempted, and the subsystem-linkage control, bit 192 of the primary ASN-second-table entry, is zero.

9. Execution of SET ADDRESS SPACE CONTROL, MOVE TO PRIMARY, or MOVE TO SECONDARY is attempted, and the secondary-space control, bit 37 of control register 0, is zero. The exception may be recognized for this reason when execution of SET ADDRESS SPACE CONTROL FAST is attempted.

10. Execution of BRANCH IN SUBSPACE GROUP is attempted, and any of the following is true:

   a. The current primary address space is not in a subspace group associated with the current dispatchable unit, that is, the primary-ASTE origin (PASTEO) in control register 5 does not equal the base-ASTE origin (BASTEO) in the dispatchable-unit control table (DUCT).

   b. The access-list-entry token (ALET) in access register Rₖ is ALET 1, but a subspace has not previously been entered by the dispatchable unit by means of BRANCH IN SUBSPACE GROUP, that is, the subspace-ASTE origin (SSASTEEO) in the DUCT is all zeros.

   c. The ALET used is other than ALET 0 and ALET 1, and the destination ASTE (DASTE) does not specify the base space or a subspace of the subspace group, that is, the DASTE origin (DASTEO) obtained from an access-list entry does not equal the BASTEO in the DUCT, and either the subspace-group bit (G) in the address-space-control element in the DASTE is zero or the base-space bit (B) in the DASTE is one.

11. Execution of BRANCH AND SET AUTHORITY is attempted, and the Rₖ field is zero in the base-authority state or nonzero in the reduced-authority state.

12. Execution of TRAP is attempted, and either of the following is true:

   a. The TRAP-enabled bit, bit 31 in bytes 44-47 of the dispatchable-unit control table, is zero.

   b. The PSW control, bit 13 of bytes 0-3 of the trap control table, is zero and bit 31 of the current PSW, the extended-addressing-mode bit, is one.

13. Execution of basic PROGRAM CALL is attempted, and the extended-addressing-mode bit, bit 31 of the current PSW, does not equal the entry-extended-addressing-mode bit, bit 129, in the entry-table entry.

14. Execution of LOAD REAL ADDRESS (LRA) is attempted in the 24-bit or 31-bit addressing mode, and bits 0-32 of the resulting real or absolute address are not all zeros.

15. Execution of the space-switching form of PROGRAM TRANSFER or SET SECONDARY ASN is attempted, the ASN-and-LX-reuse control in control register 0 is one, and the reusable-ASN bit in the located ASTE is one.

16. Execution of the space-switching form of PROGRAM TRANSFER WITH INSTANCE or SET SECONDARY ASN WITH INSTANCE is attempted, the controlled-ASN bit is one in the ASN-second-table entry used, and the CPU is in the problem state at the beginning of the operation.

The operation is suppressed.

The instruction-length code is 1, 2, or 3.
The special-operation exception is indicated by a program-interruption code of 0013 hex (or 0093 hex if a concurrent PER event is indicated).

**Specification Exception**
A specification exception is recognized when any of the following is true:

1. A one is introduced into an unassigned bit position of the PSW (that is, any of bit positions 0, 2-4, 24-30, or 33-63). This is handled as an early PSW specification exception.

2. A one is introduced into bit position 12 of the PSW. This is handled as an early PSW specification exception.

3. The PSW is invalid in any of the following ways:
   a. Bit 31 of the PSW is one and bit 32 is zero.
   b. Bits 31 and 32 of the PSW are zero, indicating the 24-bit addressing mode, and bits 64-103 of the PSW are not all zeros.
   c. Bit 31 of the PSW is zero and bit 32 is one, indicating the 31-bit addressing mode, and bits 64-96 of the PSW are not all zeros.

This is handled as an early PSW specification exception.

4. The PSW contains an odd instruction address.

5. An operand address does not designate an integral boundary in an instruction requiring such integral-boundary designation.

6. An odd-numbered general register is designated by an R field of an instruction that requires an even-numbered register designation.

7. A floating-point register other than 0, 1, 4, 5, 8, 9, 12, or 13 is designated for an extended operand.

8. The multiplier or divisor in decimal arithmetic exceeds 15 digits and sign.

9. The length of the first-operand field is less than or equal to the length of the second-operand field in decimal multiplication or division.

10. Bit positions 8-11 of MONITOR CALL do not contain zeros.

11. Bits 52 and 53 of the second-operand address of SET ADDRESS SPACE CONTROL or SET ADDRESS SPACE CONTROL FAST are not both zeros.

12. When the extended-addressing-mode bit in the PSW is zero, the basic-addressing-mode bit, bit 32, in the general register designated by the R₂ field of PROGRAM TRANSFER is zero, but bits 33-39 of the instruction address in the same register are not all zeros.

13. Execution of COMPARE AND FORM CODEWORD is attempted, and general registers 1, 2, and 3 do not initially contain even values.

14. Execution of UPDATE TREE is attempted, and the initial contents of general registers 4 and 5 are not a multiple of 8 in the 24-bit or 31-bit addressing mode or are not a multiple of 16 in the 64-bit addressing mode.

15. Execution of EXTRACT STACKED STATE is attempted, and the code in bit positions 56-63 of general register R₂ is greater than 4 when the ASN-and-LX-reuse facility is not installed or is greater than 5 when the facility is installed.

16. Execution of MOVE PAGE is attempted, and bit positions 48-51 of general register 0 do not contain zeros or bits 52 and 53 of the register are both one.

17. Execution of COMPRESSION CALL is attempted, and bits 48-51 of general register 0 have any of the values 0000 and 0110-1111 binary.

18. Execution of COMPARE LOGICAL STRING, MOVE STRING, or SEARCH STRING is attempted, and bits 32-55 of general register 0 are not all zeros.

19. Execution of EXECUTE is attempted, and the target address is odd.

20. Execution of RESUME PROGRAM is attempted, and bits 31, 32, and 64-127 of the PSW field in the second operand are not valid for placement in the current PSW. The exception is recognized if any of the following is true:
   - Bits 31 and 32 are both zero and bits 64-103 are not all zeros.
   - Bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros.
• Bits 31 and 32 are one and zero, respectively.
• Bit 127 is one.

21. Optionally if execution of LOAD PSW is attempted and bit 12 of the doubleword at the second-operand address is zero.

22. Execution of SET ADDRESSING MODE (SAM24) is attempted, and bits 0-39 of the unupdated instruction address in the PSW, bits 64-103 of the PSW, are not all zeros.

23. Execution of SET ADDRESSING MODE (SAM31) is attempted, and bits 0-32 of the unupdated instruction address in the PSW, bits 64-96 of the PSW, are not all zeros.

24. Execution of INVALIDATE DAT TABLE ENTRY is attempted, and bits 44-51 of general register $R_{basesupertwo}$ are not all zeros.

25. Execution of CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and the function code in bits 57-63 of general register 0 contain an unassigned or uninstalled function code.

26. Execution of CIPHER MESSAGE or CIPHER MESSAGE WITH CHAINING is attempted, and the $R_{basesuperone}$ or $R_{basesupertwo}$ field designates an odd-numbered register or general register 0.

27. Execution of COMPUTE INTERMEDIATE MESSAGE DIGEST, COMPUTE LAST MESSAGE DIGEST, or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and the $R_z$ field designates an odd-numbered register or general register 0.

28. Execution of CIPHER MESSAGE, CIPHER MESSAGE WITH CHAINING, COMPUTE INTERMEDIATE MESSAGE DIGEST or COMPUTE MESSAGE AUTHENTICATION CODE is attempted, and the second operand length is not a multiple of the data block size of the designated function. This specification-exception condition does not apply to the query functions.

29. Execution of SEARCH STRING UNICODE is attempted, and bits 32-47 of general register 0 are not all zeros.

The execution of the instruction identified by the old PSW is suppressed. However, for early PSW specification exceptions (causes 1-3) the operation that introduces the new PSW is completed, but an interruption occurs immediately thereafter.

Except as noted below, the instruction-length code (ILC) is 1, 2, or 3, indicating the length of the instruction causing the exception.

When the instruction address is odd (cause 4 on page 6-33), it is unpredictable whether the ILC is 1, 2, or 3.

When the exception is recognized because of an early PSW specification exception (causes 1-3) and the exception has been introduced by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, or an interruption, the ILC is 0. When the exception is introduced by SET ADDRESSING MODE (SAM24, SAM31), the ILC is 1, or it is 2 if SET ADDRESSING MODE was the target of EXECUTE. When the exception is introduced by SET SYSTEM MASK or by STORE THEN OR SYSTEM MASK, the ILC is 2.

The specification exception is indicated by a program-interruption code of 0006 hex (or 0086 hex if a concurrent PER event is indicated).

**Programming Note:** See "Exceptions Associated with the PSW" on page 6-9 for a definition of when the exceptions associated with the PSW are recognized.

### Stack-Empty Exception

A stack-empty exception is recognized during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN when the current linkage-stack entry is a header entry and the backward stack-entry validity bit in the header entry is zero.

The operation is nullified.

The instruction-length code is 1 or 2.

The stack-empty exception is indicated by a program-interruption code of 0031 hex (or 00B1 hex if a concurrent PER event is indicated).
Stack-Full Exception
A stack-full exception is recognized during the stacking process in BRANCH AND STACK or stacking PROGRAM CALL when there is not enough remaining free space in the current linkage-stack section and the forward-section validity bit in the trailer entry of the section is zero.

The operation is nullified.

The instruction-length code is 2.

The stack-full exception is indicated by a program-interruption code of 0030 hex (or 00B0 hex if a concurrent PER event is indicated).

Stack-Operation Exception
A stack-operation exception is recognized during the unstacking process in PROGRAM RETURN when the unstack-suppression bit is one in any linkage-stack state entry or header entry encountered during the process.

The operation is nullified.

The instruction-length code is 1 or 2.

The stack-operation exception is indicated by a program-interruption code of 0034 hex (or 00B4 hex if a concurrent PER event is indicated).

Stack-Specification Exception
A stack-specification exception is recognized in each of the following cases:
1. During the stacking process in BRANCH AND STACK or stacking PROGRAM CALL when there is not enough remaining free space in the current linkage-stack section and either of the following is true:
   a. The remaining-free-space value used to locate the trailer entry of the current section is not a multiple of 8.
   b. There is not enough remaining free space in the next section.
2. During the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN when the current linkage-stack entry is a header entry in which the backward stack-entry address designates another header entry.

The operation is nullified.

The instruction-length code is 1 or 2.

The stack-specification exception is indicated by a program-interruption code of 0032 hex (or 00B2 hex if a concurrent PER event is indicated).

Stack-Type Exception
A stack-type exception is recognized during the unstacking process in EXTRACT STACKED REGISTERS, EXTRACT STACKED STATE, MODIFY STACKED STATE, or PROGRAM RETURN in each of the following cases:
1. The current linkage-stack entry is not a header entry or a state entry.
2. When the current linkage-stack entry is a header entry, the preceding entry, designated by the backward stack-entry address in the header entry, is not a header entry or a state entry. (A stack-specification exception is recognized if the preceding entry is a header entry.)

The operation is nullified.

The instruction-length code is 1 or 2.

The stack-type exception is indicated by a program-interruption code of 0033 hex (or 00B3 hex if a concurrent PER event is indicated).

Trace-Table Exception
A trace-table exception is recognized when the CPU attempts to store a trace-table entry which would reach or cross the next 4K-byte block boundary. For the purpose of recognizing this exception in the TRACE instruction, the explicit trace entry is treated as being 76 bytes long for TRACE (TRACE) and as 144 bytes long for TRACE (TRACG). For a PROGRAM CALL instruction that would cause storing of both a PROGRAM CALL trace entry and a mode-switch trace entry, the exception is recognized for the first entry when either the first or the second entry would reach or cross the boundary.

The operation is nullified.

The instruction-length code is 1, 2, or 3, indicating the length of the instruction causing the exception.

The trace-table exception is indicated by a program-interruption code of 0016 hex (or 0096 hex if a concurrent PER event is indicated).
Translation-Specification Exception

A translation-specification exception is recognized when translation of a virtual address is attempted and any of the following is true:

1. In the lookup in the table designated by the address-space-control element used for the translation, the table-type bits in the selected table entry do not equal the designation-type bits in the address-space-control element.

2. In a lookup in a table designated by an entry in a region first table, region second table, or region third table, the value of the table-type bits in the selected table entry is not one less than the value of the same bits in the designating table entry.

3. The private-space control, bit 55 in the address-space-control element used for the translation, is one, the segment-table entry used for the translation is valid, and the common-segment bit, bit 59, in the segment-table entry is one.

4. The page-table entry used for the translation is valid, and bit positions 52 and 55 in the entry do not contain zeros.

Any of the above reasons is referred to by saying that the DAT-table entry has a format error.

The exception is recognized only as part of the execution of an instruction using address translation, that is, when DAT is on and a logical address, instruction address, or virtual address must be translated, or when LOAD REAL ADDRESS or STORE REAL ADDRESS is executed.

The unit of operation is suppressed.

When the exception occurs during fetching of an instruction, it is unpredictable whether the ILC is 1, 2, or 3. When the exception occurs during the fetching of the target of EXECUTE, the ILC is 2.

When the exception occurs during a reference to an operand location, the instruction-length code (ILC) is 1, 2, or 3 and indicates the length of the instruction causing the exception.

The translation-specification exception is indicated by a program-interruption code of 0012 hex (or 0092 hex if a concurrent PER event is indicated).

Programming Note: When a translation-specification exception is recognized in the process of translating an instruction address, the operation is suppressed. In this case, the instruction-length code (ILC) is needed to derive the address of the instruction, as the instruction address in the old PSW has been incremented by the amount indicated by the ILC. In the case of region-first-translation, region-second-translation, region-third-translation, segment-translation, and page-translation exceptions, the operation is nullified, the instruction address in the old PSW identifies the instruction, and the ILC may be arbitrarily set to 1, 2, or 3.

Collective Program-Interruption Names

For the sake of convenience, certain program exceptions are grouped together under a single collective name. These collective names are used when it is necessary to refer to the complete set of exceptions, such as in instruction definitions. Four collective names are used:

- Access exceptions
- ASN-translation exceptions
- Subspace-replacement exceptions
- Trace exceptions

The individual exceptions and their priorities are listed in “Multiple Program-Interruption Conditions” on page 6-39.

Recognition of Access Exceptions

Figure 6-5 on page 6-37 summarizes the conditions that can cause access exceptions and the action taken when they are encountered.
<table>
<thead>
<tr>
<th>Condition</th>
<th>Indication</th>
<th>Action</th>
<th>Indication</th>
<th>Action</th>
<th>Indication</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access register</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>AS</td>
<td>Suppress</td>
</tr>
<tr>
<td>Bits 0-6 not all zeros</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effective access-list designation</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
</tr>
<tr>
<td>Invalid address of designation</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>AT</td>
<td>Nullify</td>
</tr>
<tr>
<td>Access-list entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access-list-length violation</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
</tr>
<tr>
<td>I bit on</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>AT</td>
<td>Nullify</td>
</tr>
<tr>
<td>Sequence number in access register not equal to sequence number in entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASN-second-table entry</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
</tr>
<tr>
<td>Invalid address of entry</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>AS</td>
<td>Nullify</td>
</tr>
<tr>
<td>I bit on</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>AS</td>
<td>Nullify</td>
</tr>
<tr>
<td>Authority-table entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authority-table-length violation</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>EA</td>
<td>Nullify</td>
</tr>
<tr>
<td>Invalid address of entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secondary-authority bit not one</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>EA</td>
<td>Nullify</td>
</tr>
<tr>
<td>Address-space-control element</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits 0-30, 0-21, or 0-32 of instruction or operand address not all zeros when address-space-control element is a region-second-table designation, region-third-table designation, or segment-table designation, respectively</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>ATY</td>
<td>Nullify</td>
</tr>
<tr>
<td>Region-table-entry designated by address-space-control element or region-table-entry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry outside of table</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>RT</td>
<td>Nullify</td>
</tr>
<tr>
<td>Invalid address of entry</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
<td>A</td>
<td>Suppress</td>
</tr>
<tr>
<td>I bit on</td>
<td>cc3</td>
<td>Complete</td>
<td>cc3</td>
<td>Complete</td>
<td>RT</td>
<td>Nullify</td>
</tr>
<tr>
<td>TT in entry not equal DT in address-space-control element or not one less than TT in next-higher-level entry</td>
<td>T5</td>
<td>Suppress</td>
<td>T5</td>
<td>Suppress</td>
<td>T5</td>
<td>Suppress</td>
</tr>
</tbody>
</table>

Figure 6-5 (Part 1 of 3). Handling of Access Exceptions
<table>
<thead>
<tr>
<th>Condition 2</th>
<th>Translation for Virtual Address of LRA or LRAG</th>
<th>Translation for TAR and TPROT, and Access for Logical Address of TPROT 1</th>
<th>Translation and Access for Any Other Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indication</td>
<td>Action</td>
<td>Indication</td>
<td>Action</td>
</tr>
</tbody>
</table>

**Segment-table entry designated by address-space-control element or region-table entry**
- Entry outside of Table: cc3 Complete cc3 Complete ST Nullify
- Invalid address of Table: A Suppress A Suppress A Suppress
- I bit on (except as follows): cc1 Complete cc3 Complete ST Nullify
- I bit on (LRA in 24-bit or 31-bit mode when bits 0-32 of entry address not all zeros): cc3 Complete
- One in a bit position which is checked for zero: TS Suppress TS Suppress TS Suppress

**Page-table entry**
- Invalid address of entry: A Suppress A Suppress A Suppress
- I bit on (except as follows): cc2 Complete cc3 Complete PT Nullify
- I bit on (LRA in 24-bit or 31-bit mode when bits 0-32 of entry address not all zeros): cc3 Complete
- One in a bit position which is checked for zero: TS Suppress TS Suppress TS Suppress

**Access for instruction fetch**
- Location protected (key-controlled protection): - - - - P Suppress
- Invalid address: - - - A Suppress

**Access for operands**
- Location protected (low-address, page, or key-controlled protection): - - cc set 4 Complete P Term.
- Invalid address: - - A Suppress A Term.

**Explanation:**
- The condition does not apply.
- 4 Action is to terminate except where otherwise specified in this publication. For access-list-controlled protection and page protection, the action is always to suppress.
- TAR does not have a logical address. The rows "Address-space-control element" through "Access for operands" apply only to TPROT, not to TAR.
- Protection applies only to accesses for instruction fetch and for operands. It does not apply to the fetching of the effective access-list designation or any of the listed entries.
- Exceptions related to an access register, effective access-list designation, access-list entry, ASN-second-table entry, or authority-table entry are recognized only in the access-register mode except that, for LOAD REAL ADDRESS and STORE REAL ADDRESS, they are recognized when PSW bits 16 and 17 are 01 binary, and, for TEST ACCESS, they are recognized regardless of the translation mode.
- Authority table is not accessed and secondary-authority bit is not checked if the private bit in the access-list entry is zero or the access-list-authorization index in the access-list entry is equal to the extended authorization index in control register B.
- A translation-specification exception for a format error in a table entry is recognized only when the execution of an instruction requires the entry for translation of an address.

Figure 6-5 (Part 2 of 3). Handling of Access Exceptions
Any access exception is recognized as part of the execution of the instruction with which the exception is associated. An access exception is not recognized when the CPU attempts to prefetch from an unavailable location or detects some other access-exception condition, but a branch instruction or an interruption changes the instruction sequence such that the instruction is not executed.

Every instruction can cause an access exception to be recognized because of instruction fetch. Additionally, access exceptions associated with instruction execution may occur because of an access to an operand in storage.

An access exception due to fetching an instruction is indicated when the first instruction halfword cannot be fetched without encountering the exception. When the first halfword of the instruction has no access exceptions, access exceptions may be indicated for additional halfwords according to the instruction length specified by the first two bits of the instruction; however, when the operation can be performed without accessing the second or third halfwords of the instruction, it is unpredictable whether the access exception is indicated for the unused part. Since the indication of access exceptions for instruction fetch is common to all instructions, it is not covered in the individual instruction definitions.

Except where otherwise indicated in the individual instruction description, the following rules apply for exceptions associated with an access to an operand location. For a fetch-type operand, access exceptions are necessarily indicated only for that portion of the operand which is required for completing the operation. It is unpredictable whether access exceptions are indicated for those portions of a fetch-type operand which are not required for completing the operation. For a store-type operand, access exceptions are recognized for the entire operand even if the operation could be completed without the use of the inaccessible part of the operand. In situations where the value of a store-type operand is defined to be unpredictable, it is unpredictable whether an access exception is indicated.

Whenever an access to an operand location can cause an access exception to be recognized, the word “access” is included in the list of program exceptions in the description of the instruction. This entry also indicates which operand can cause the exception to be recognized and whether the exception is recognized on a fetch or store access to that operand location. Access exceptions are recognized only for the portion of the operand as defined for each particular instruction.

### Multiple Program-Interruption Conditions

Except for PER events, only one program-interruption condition is indicated with a program interruption. The existence of one condition, however, does not preclude the existence of other conditions. When more than one program-
interruption condition exists, only the condition having the highest priority is identified in the interruption code.

With two conditions of the same priority, it is unpredictable which is indicated. In particular, the priority of access exceptions associated with the two parts of an operand that crosses a page or protection boundary is unpredictable and is not necessarily related to the sequence specified for the access of bytes within the operand.

The type of ending which occurs (nullification, suppression, or termination) is that which is defined for the type of exception that is indicated in the interruption code. However, if a condition is indicated which permits termination, and another condition also exists which would cause either nullification or suppression, then the unit of operation is suppressed.

The relative priorities of any two conditions listed in the figure can be found by comparing the priority numbers, as found in the figure, from left to right until a mismatch is found. If the first inequality is between numeric characters, either the two conditions are mutually exclusive or, if both can occur, the condition with the smaller number is indicated. If the first inequality is between alphabetic characters, then the two conditions are not exclusive, and it is unpredictable which is indicated when both occur.

To understand the use of the table, consider an example involving the instruction ADD DECIMAL, which is a six-byte instruction. Assume that the first four bytes of the instruction can be accessed but that the instruction crosses a boundary so that an addressing exception exists for the last two bytes. Additionally, assume that the first operand addressed by the instruction contains invalid decimal digits and is in a location that can be fetched from, but not stored into, because of key-controlled protection. The three exceptions which could result from attempted execution of the ADD DECIMAL are:

<table>
<thead>
<tr>
<th>Priority Number</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.B</td>
<td>Access exceptions (operand 1).</td>
</tr>
<tr>
<td>8.D</td>
<td>Data exception.</td>
</tr>
</tbody>
</table>

Since the first inequality (7≠8) is between numeric characters, the addressing exception would be indicated. If, however, the entire ADD DECIMAL instruction can be fetched, and only the second two exceptions listed above exist, then the inequality (B≠D) is between alphabetic characters, and it is unpredictable whether the protection exception or the data exception would be indicated.

For some instructions, the priority is shown in the individual instruction description.

Figure 6-6 on page 6-41 lists the priorities of all program-interruption conditions other than PER events and exceptions associated with some of the more complex control instructions. All exceptions associated with references to storage for a particular instruction halfword or a particular operand byte are grouped as a single entry called “access.” Figure 6-7 on page 6-44 lists the priority of access exceptions for a single access. Thus, the second figure specifies which of several exceptions, encountered either in the access of a particular portion of an instruction or in any particular access associated with an operand, has highest priority, and the first figure specifies the priority of this condition in relation to other conditions detected in the operation. Similarly, the priorities for exceptions occurring as part of ASN translation and tracing are covered in Figure 6-8 on page 6-46 and Figure 6-10 on page 6-47, respectively.
1. Specification exception due to any PSW error of the type that causes an immediate interruption.¹

2. Specification exception due to an odd instruction address in the PSW.

3. Access exceptions for first halfword of EXECUTE.²

4. Access exceptions for second halfword of EXECUTE.²

5. Specification exception due to target instruction of EXECUTE not being specified on halfword boundary.²


7.A Access exceptions for second instruction halfword.³

7.B Access exceptions for third instruction halfword.³

7.C.1 Operation exception.

7.C.2 Privileged-operation exception for privileged instructions.

7.C.3 Execute exception.

7.C.4 Special-operation exception.⁴

8.A Specification exception due to conditions other than those included in 1, 2, and 5 above.

8.B⁵ Access exceptions for an access to an operand in storage.⁶

8.C⁵ Access exceptions for any other access to an operand in storage.⁶

8.D Data exception.⁷

8.E Decimal-divide exception.⁸

8.F Trace exceptions.

9. Events other than PER events, exceptions which result in completion, and the following exceptions: fixed-point divide, floating-point divide, operand, square root, and unnormalized operand. Either these exceptions and events are mutually exclusive or their priority is specified in the corresponding definitions.
Explanation:

Numbers indicate priority, with "1" being the highest priority; letters indicate no priority.

1 PSW errors which cause an immediate interruption may be introduced by a new PSW loaded as a result of an interruption or by the instructions LOAD PSW, PROGRAM RETURN, SET SYSTEM MASK, and STORE THEN OR SYSTEM MASK. The priority shown in the chart is for a PSW error introduced by an interruption and may also be considered as the priority for a PSW error introduced by the previous instruction. The error is introduced only if the instruction encounters no other exceptions. The resulting interruption has a higher priority than any interruption caused by the instruction which would have been executed next; it has lower priority, however, than any interruption caused by the instruction which introduced the erroneous PSW.

2 Priorities 3, 4, and 5 are for the EXECUTE instruction, and priorities starting with 6 are for the target instruction. When no EXECUTE is encountered, priorities 3, 4, and 5 do not apply.

3 Separate accesses may occur for each halfword of an instruction. The second instruction halfword is accessed only if bits 0-1 of the instruction are not both zeros. The third instruction halfword is accessed only if bits 0-1 of the instruction are both ones. Access exceptions for one of these halfwords are not necessarily recognized if the instruction can be completed without use of the contents of the halfword or if an exception of lower priority can be determined without the use of the halfword.

4 The special-operation exception recognized by LOAD REAL ADDRESS has priority 9. See the definition of the TRAP instruction for other priorities of the special-operation exception.

5 As in instruction fetching, separate accesses may occur for each portion of an operand. Each of these accesses, and also accesses for different operands, are of equal priority, and the two entries 8.B and 8.C are listed to represent the relative priorities of exceptions associated with any two of these accesses. Access exceptions for INSERT STORAGE KEY EXTENDED, INSERT VIRTUAL STORAGE KEY, INVALIDATE PAGE TABLE ENTRY, LOAD REAL ADDRESS, STORE REAL ADDRESS, RESET REFERENCE BIT EXTENDED, SET STORAGE KEY EXTENDED, and TEST PROTECTION are also included in 8.B.

6 For MOVE LONG, MOVE LONG EXTENDED, COMPARE LOGICAL LONG, and COMPARE LOGICAL LONG EXTENDED, an access exception for a particular operand can be indicated only if the R field for that operand designates an even-numbered register.

7 The exception can be indicated only if the sign, digit, or digits responsible for the exception were fetched without encountering an access exception.

8 The exception can be indicated only if the digits used in establishing the exception, and also the signs, were fetched without encountering an access exception, only if the signs are valid, and only if the digits used in establishing the exception are valid.
Access Exceptions
The access exceptions consist of those exceptions which can be encountered while using an absolute, instruction, logical, real, or virtual address to access storage. Thus, in the access-register mode, the exceptions are:

1. ALET specification
2. ALEN translation
3. ALE sequence
4. ASTE validity
5. ASTE sequence
6. Extended authority
7. Addressing (the ART tables)
8. ASCE type
9. Region first translation
10. Region second translation
11. Region third translation
12. Segment translation
13. Page translation
14. Translation specification
15. Addressing (the DAT tables)
16. Addressing (the operand or instruction)
17. Protection (key-controlled, access-list-controlled, page, and low-address)

With DAT on but in other than the access-register mode, exceptions 8-17 in the above list, except for access-list-controlled protection, can be encountered.

With DAT off, the exceptions are:

1. Addressing (the operand or instruction)
2. Protection (key-controlled and low-address)

Additionally, even with DAT off, the instruction STORE REAL ADDRESS can encounter exceptions 1-17, the instruction LOAD REAL ADDRESS can encounter exceptions 7, 14, and 15, and the instruction INVALIDATE PAGE TABLE ENTRY can encounter exception 15.

The access exceptions are listed in more detail in Figure 6-7 on page 6-44.

Programming Note: The priorities in Figure 6-7 on page 6-44 could be renumbered, but they are kept as they are to allow easier comparison to the corresponding ESA/390 priorities. Specifically, B.1.A.1-B.1.A.9 could be changed to B.1-B.9, but ESA/390 contains “B.1.B Translation-specification exception due to invalid encoding of bits 8-12 of control register 0.”
| A. Protection exception (low-address protection) due to a store-type operand reference with an effective address in the range 0-511 or 4096-4607. Not recognized if DAT is on and the address-space-control element to be used in the translation cannot be obtained because of another exception. |
| B.1.A.1 ALET-specification exception due to bits 0-6 of access register not being all zeros.¹ |
| B.1.A.2 Addressing exception for access to effective access-list designation.² |
| B.1.A.3 ALEN-translation exception due to access-list entry being outside the list.¹ |
| B.1.A.4 Addressing exception for access to access-list entry.² |
| B.1.A.5 ALEN-translation exception due to I bit in access-list entry having the value one.¹ |
| B.1.A.6 ALE-sequence exception due to access-list-entry sequence number (ALESN) in access register not being equal to ALESN in access-list entry.¹ |
| B.1.A.7 Addressing exception for access to ASN-second-table entry.² |
| B.1.A.8 ASTE-validity exception due to I bit in ASN-second-table entry having the value one.¹ |
| B.1.A.9 ASTE-sequence exception due to ASN-second-table-entry sequence number (ASTESN) in access-list entry not being equal to ASTESN in ASN-second-table entry.¹ |

Note: Exceptions B.1.A.10 through B.1.A.12 are recognized only when the private bit in the access-list entry is one and the ALEAX in the entry is not equal to the EAX in control register 8.

| B.1.A.10 Extended-authority exception due to authority-table entry being outside table.¹ |
| B.1.A.11 Addressing exception for access to authority-table entry.² |
| B.1.A.12 Extended-authority exception due to (1) private bit in access-list entry not being zero, (2) access-list-entry authorization index in access-list entry not being equal to extended authorization index in control register 8, and (3) secondary-authority bit selected by extended authorization index not being one.¹ |

Figure 6-7 (Part 1 of 3). Priority of Access Exceptions
B.2.A Protection exception (access-list-controlled protection) due to store-type operand reference to a virtual address which is protected against stores.¹

B.2.B.1 ASCE-type exception due to bits 0-10, 0-21, or 0-32 of instruction or operand address not being zeros when address-space-control element is a region-second-table designation, region-third-table designation, or segment-table designation, respectively.³

B.2.B.2 Region-first-, region-second-, region-third-, or segment-translation exception due to required entry in table designated by address-space-control element being outside of table.³

Note: Exceptions B.2.B.3 through B.2.B.6 are recognized for a region-first-table, region-second-table, region-third-table, and segment-table entry in the order in which the entries are used.

B.2.B.3 Addressing exception for access to table entry.⁴

B.2.B.4 Region-first-, region-second-, region-third-, or segment-translation exception due to I bit in table entry having the value one.³

B.2.B.5 Translation-specification exception due to (1) TT in table entry not equal to DT in designating address-space-control element or not one less than TT in designating next-higher-level table entry or (2) invalid one in segment-table entry if this entry is a segment-table entry (common-segment bit if private-space bit in address-space-control element is one).⁴

B.2.B.6 Region-second-, region-third-, or segment-translation exception due to required entry in next-lower-level table entry, if any, being outside of table.³

B.2.B.7 Addressing exception for access to page-table entry.⁵

B.2.B.8 Page-translation exception due to I bit in page-table entry having the value one.³ ⁷

B.2.B.9 Translation-specification exception due to invalid ones in page-table entry (bits 52 and 55) in which I bit is zero.⁴

B.3.A Protection exception (page protection) due to a store-type operand reference to a virtual address which is protected against stores.⁶

Figure 6-7 (Part 2 of 3). Priority of Access Exceptions
B.3.B Addressing exception for access to instruction or operand.

B.4. Protection exception (key-controlled protection) due to attempt to access a protected instruction or operand location.

Explanation:

1. Not applicable when not in the access-register mode; not applicable for execution of TEST ACCESS and for translation of operand address of LOAD REAL ADDRESS and TEST PROTECTION.

2. Not applicable when not in the access-register mode, except applicable for execution of TEST ACCESS and, when PSW bits 16 and 17 are 01 binary, for translation of operand address of LOAD REAL ADDRESS and second-operand address of STORE REAL ADDRESS.

3. Not applicable when DAT is off except for translation of second-operand address of STORE REAL ADDRESS; not applicable to operand addresses of LOAD REAL ADDRESS and TEST PROTECTION.

4. Not applicable when DAT is off except for translation of operand address of LOAD REAL ADDRESS and second-operand address of STORE REAL ADDRESS.

5. Not applicable when DAT is off, except for execution of INVALIDATE PAGE TABLE ENTRY and for translation of operand address of LOAD REAL ADDRESS and second-operand address of STORE REAL ADDRESS.

6. Not applicable when DAT is off.

7. For MOVE PAGE, if the condition is true for both operands, the exception is recognized for the second operand. Also, if the condition-code-option bit is one, the exception is not recognized. Instead, condition code 1 is set if the condition is true for only the first operand, or condition code 2 is set if the condition is true for the second operand or both operands.

Figure 6-7 (Part 3 of 3). Priority of Access Exceptions

ASN-Translation Exceptions

The ASN-translation exceptions are those exceptions which are common to the process of translating an ASN in the instructions PROGRAM RETURN, PROGRAM TRANSFER, and SET SECONDARY ASN. The exceptions and the priority in which they are detected are shown in Figure 6-8.

1. Addressing exception for access to ASN-first-table entry.

2. AFX-translation exception due to I bit (bit 0) in ASN-first-table entry being one.

3. Addressing exception for access to ASN-second-table entry.

4. ASX-translation exception due to I bit (bit 0) in ASN-second-table entry being one.

Figure 6-8. Priority of ASN-Translation Exceptions
Subspace-Replacement Exceptions

The subspace-replacement exceptions are those exceptions which can be recognized during a subspace-replacement operation in PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE. The exceptions and their priority are shown in Figure 6-9 on page 6-47.

| 1. Addressing exception for access to dispatchable-unit control table. |
| 2. Addressing exception for access to subspace ASN-second-table entry. |
| 3. ASTE-validity exception due to bit 0 being one in subspace ASN-second-table entry. |
| 4. ASTE-sequence exception due to subspace ASN-second-table-entry sequence number in dispatchable-unit control table not being equal to ASN-second-table-entry sequence number in subspace ASN-second-table entry. |

Figure 6-9. Priority of Subspace-Replacement Exceptions

Trace Exceptions

The trace exceptions are those exceptions which can be encountered while forming a trace-table entry. The exceptions and their priority are shown in Figure 6-10.

A. Protection exception (low-address protection) due to entry address being in the range 0-511 or 4096-4607.

B.1 Trace-table exception due to new entry reaching or crossing next 4K-byte boundary.

B.2 Addressing exception for access to trace-table entry.

Figure 6-10. Priority of Trace Exceptions

Restart Interruption

The restart interruption provides a means for the operator or another CPU to invoke the execution of a specified program. The CPU cannot be disabled for this interruption.

A restart interruption causes the old PSW to be stored at real locations 288-303 and a new PSW, designating the start of the program to be executed, to be fetched from real locations 416-431. The instruction-length code and interruption code are not stored.

If the CPU is in the operating state, the exchange of the PSWs occurs at the completion of the current unit of operation and after all other pending interruption conditions for which the CPU is enabled have been honored. If the CPU is in the stopped state, the CPU enters the operating state and exchanges the PSWs without first honoring any other pending interruptions.

The restart interruption is initiated by activating the restart key. The operation can also be initiated at the addressed CPU by executing a SIGNAL PROCESSOR instruction which specifies the restart order.

When the rate control is set to the instruction-step position, it is unpredictable whether restart causes a unit of operation or additional interruptions to be performed after the PSWs have been exchanged.

Programming Note: To perform a restart when the CPU is in the check-stop state, the CPU has to be reset. Resetting with loss of the least amount of information can be accomplished by means of the system-reset-normal key, which does not clear the contents of program-addressable registers, including the control registers, but causes the channel subsystem to be reset. The CPU-reset SIGNAL PROCESSOR order can be used to clear the CPU without affecting the channel subsystem.

Supervisor-Call Interruption

The supervisor-call interruption occurs when the instruction SUPERVISOR CALL is executed. The CPU cannot be disabled for the interruption, and the interruption occurs immediately upon the execution of the instruction.

The supervisor-call interruption causes the old PSW to be stored at real locations 320-335 and a new PSW to be fetched from real locations 448-463.

The contents of bit positions 8-15 of the SUPERVISOR CALL instruction are placed in the right-most byte of the interruption code. The leftmost
byte of the interruption code is set to zero. The instruction-length code is 1, unless the instruction was executed by means of EXECUTE, in which case the code is 2.

The interruption code is placed at real locations 138-139; the instruction-length code is placed in bit positions 5 and 6 of the byte at real location 137, with the other bits set to zeros; and zeros are stored at real location 136.

Priority of Interruptions

During the execution of an instruction, several interruption-causing events may occur simultaneously. The instruction may give rise to a program interruption, a request for an external interruption may be received, equipment malfunctioning may be detected, an I/O-interruption request may be made, and the restart key may be activated. Instead of the program interruption, a supervisor-call interruption might occur; or both can occur if PER is active. Simultaneous interruption requests are honored in a predetermined order.

An exigent machine-check condition has the highest priority. When it occurs, the current operation is terminated or nullified. Program and supervisor-call interruptions that would have occurred as a result of the current operation may be eliminated. Any pending repressible machine-check conditions may be indicated with the exigent machine-check interruption. Every reasonable attempt is made to limit the side effects of an exigent machine-check condition, and requests for external, I/O, and restart interruptions normally remain unaffected.

In the absence of an exigent machine-check condition, interruption requests existing concurrently at the end of a unit of operation are honored, in descending order of priority, as follows:

- Supervisor call
- Program
- Repressible machine check
- External
- Input/output
- Restart

The processing of multiple simultaneous interruption requests consists in storing the old PSW and fetching the new PSW belonging to the interruption first honored. This new PSW is subsequently stored without the execution of any instructions, and the new PSW associated with the next interruption is fetched. Storing and fetching of PSWs continues until no more interruptions are to be serviced. The priority is reevaluated after each new PSW is loaded. Each evaluation takes into consideration any additional interruptions which may have become pending. Additionally, external and I/O interruptions, as well as machine-check interruptions due to repressible conditions, occur only if the current PSW at the instant of evaluation indicates that the CPU is interruptible for the cause.

Instruction execution is resumed using the last-fetched PSW. The order of executing interruption subroutines is, therefore, the reverse of the order in which the PSWs are fetched.

If the new PSW for a program interruption does not specify the wait state and has an odd instruction address, or causes an access exception to be recognized, another program interruption occurs. Since this second interruption introduces the same unacceptable PSW, a string of interruptions is established. These program exceptions are recognized as part of the execution of the following instruction, and the string may be broken by an external, I/O, machine-check, or restart interruption or by the stop function.

If the new PSW for a program interruption contains a one in bit position 12 or in an unassigned bit position, if the leftmost 40 bits of the instruction address are not zeros when bit 31 and 32 indicate 24-bit addressing, or the leftmost 33 bits are not zeros when bits 31 and 32 indicate 31-bit addressing, or if bit 32 is zero when bit 31 is one, another program interruption occurs. This condition is of higher priority than restart, I/O, external, or repressible machine-check conditions, or the stop function, and CPU reset has to be used to break the string of interruptions.

A string of interruptions for other interruption classes can also exist if the new PSW allows the interruption which has just occurred. These include machine-check interruptions, external interruptions, and I/O interruptions due to PCI conditions generated because of CCWs which form a loop. Furthermore, a string of interruptions involving more than one interruption class can exist. For example, assume that the CPU timer is negative and the CPU-timer subclass mask is one. If the external new PSW has a one in an unas-
signed bit position, and the program new PSW is enabled for external interruptions, then a string of interruptions occurs, alternating between external and program. Even more complex strings of interruptions are possible. As long as more interruptions must be serviced, the string of interruptions cannot be broken by employing the stop function; CPU reset is required.

Similarly, CPU reset has to be invoked to terminate the condition that exists when an interruption is attempted with a prefix value designating a storage location that is not available to the CPU.

Interruptions for all requests for which the CPU is enabled occur before the CPU is placed in the stopped state. When the CPU is in the stopped state, restart has the highest priority.

Programming Note: The order in which concurrent interruption requests are honored can be changed to some extent by masking.
Chapter 7. General Instructions

Data Format ........................................... 7-2
Binary-Integer Representation ................. 7-3
Binary Arithmetic ................................. 7-4
  Signed Binary Arithmetic ...................... 7-4
    Addition and Subtraction .................. 7-4
    Fixed-Point Overflow ...................... 7-4
  Unsigned Binary Arithmetic ................. 7-5
Signed and Logical Comparison .................. 7-5
Instructions ....................................... 7-6
  ADD ........................................... 7-18
  ADD HALFWORD ................................ 7-18
  ADD HALFWORD IMMEDIATE .................... 7-18
  ADD LOGICAL ................................ 7-19
  ADD LOGICAL WITH CARRY .................... 7-20
  AND ........................................ 7-20
  AND IMMEDIATE ................................ 7-21
  BRANCH AND LINK ................................ 7-22
  BRANCH AND SAVE ................................ 7-23
  BRANCH AND SAVE AND SET MODE ............ 7-23
  BRANCH AND SET MODE ......................... 7-24
  BRANCH ON CONDITION ......................... 7-25
  BRANCH ON COUNT ................................ 7-26
  BRANCH ON INDEX HIGH ....................... 7-27
  BRANCH ON INDEX LOW OR EQUAL ............ 7-27
  BRANCH RELATIVE AND SAVE .................. 7-28
  BRANCH RELATIVE AND SAVE LONG ........... 7-28
  BRANCH RELATIVE ON CONDITION ............. 7-29
  BRANCH RELATIVE ON CONDITION LONG ........ 7-29
  BRANCH RELATIVE ON COUNT .................. 7-29
  BRANCH RELATIVE ON INDEX HIGH ............ 7-30
  BRANCH RELATIVE ON INDEX LOW OR EQUAL .. 7-30
CHECKSUM ......................................... 7-31
CIPHER MESSAGE (KM) ............................. 7-35
CIPHER MESSAGE WITH CHAINING (KMC) .......... 7-35
COMPARE ........................................... 7-45
COMPARE AND FORM CODEWORD .................. 7-46
COMPARE AND SWAP ................................ 7-53
COMPARE DOUBLE AND SWAP ..................... 7-53
COMPARE HALFWORD ................................ 7-55
COMPARE HALFWORD IMMEDIATE ................. 7-55
COMPARE LOGICAL ................................ 7-56
COMPARE LOGICAL CHARACTERS UNDER MASK .... 7-57
COMPARE LOGICAL LONG ......................... 7-58
COMPARE LOGICAL LONG EXTENDED ............. 7-60
COMPARE LOGICAL LONG UNICODE .............. 7-64
COMPARE LOGICAL STRING ...................... 7-67
COMPARE UNTIL SUBSTRING EQUAL .............. 7-68
COMPRESSION CALL ............................... 7-72
COMPUTE INTERMEDIATE MESSAGE DIGEST (KIMD) 7-84
COMPUTE LAST MESSAGE DIGEST (KLMD) .......... 7-84
COMPUTE MESSAGE AUTHENTICATION CODE (KMAC) 7-91
CONVERT TO BINARY .............................. 7-97
CONVERT TO DECIMAL ............................ 7-98
CONVERT UTF-16 TO UTF-32 ..................... 7-98
CONVERT UTF-16 TO UTF-8 ..................... 7-101
CONVERT UNICODE TO UTF-8 .................... 7-101
CONVERT UTF-32 TO UTF-16 .................... 7-104
CONVERT UTF-32 TO UTF-8 ..................... 7-107
CONVERT UTF-8 TO UTF-16 ..................... 7-110
CONVERT UTF-8 TO UNICODE .................... 7-110
CONVERT UTF-8 TO UTF-32 ..................... 7-113
COPY ACCESS ..................................... 7-116
DIVIDE .......................................... 7-116
DIVIDE LOGICAL .................................. 7-117
DIVIDE SINGLE .................................. 7-117
EXCLUSIVE OR .................................... 7-118
EXECUTE ......................................... 7-119
EXTRACT ACCESS .................................. 7-120
EXTRACT PSW ...................................... 7-120
INSERT CHARACTER .............................. 7-121
INSERT CHARACTERS UNDER MASK ............. 7-121
INSERT IMMEDIATE ................................ 7-122
INSERT PROGRAM MASK ........................... 7-122
LOAD ............................................. 7-123
LOAD ACCESS MULTIPLE ......................... 7-123
LOAD ADDRESS .................................... 7-124
LOAD ADDRESS EXTENDED ....................... 7-124
LOAD ADDRESS RELATIVE LONG .................. 7-125
LOAD AND TEST ................................... 7-126
LOAD BYTE ....................................... 7-126
LOAD COMPLEMENT ................................ 7-126
LOAD HALFWORD .................................. 7-127
LOAD HALFWORD IMMEDIATE ..................... 7-127
LOAD LOGICAL .................................... 7-128
LOAD LOGICAL CHARACTER ..................... 7-128
LOAD LOGICAL HALFWORD ....................... 7-128
LOAD LOGICAL IMMEDIATE ...................... 7-128
LOAD LOGICAL THIRTY ONE BITS ............... 7-129
LOAD MULTIPLE .................................. 7-129
LOAD MULTIPLE DISJOINT ...................... 7-130
LOAD MULTIPLE HIGH ............................. 7-130
LOAD NEGATIVE ................................... 7-130
LOAD PAIR FROM QUADWORD ..................... 7-131

© Copyright IBM Corp. 1990-2004
This chapter includes all the unprivileged instructions described in this publication other than the decimal and floating-point instructions.

### Data Format

The general instructions treat data as being of four types: signed binary integers, unsigned binary integers, unstructured logical data, and decimal data. Data is treated as decimal by the conversion, packing, and unpacking instructions. Decimal data is described in [Chapter 8, “Decimal Instructions.”](#)

The general instructions manipulate data which resides in general registers or is introduced from the instruction stream. Some general instructions operate on data which resides in the PSW or the TOD clock.

In a storage-and-storage operation the operand fields may be defined in such a way that they overlap. The effect of this overlap depends upon the operation. When the operands remain unchanged, as in COMPARE or TRANSLATE AND TEST, overlapping does not affect the execution of the operation. For instructions such as MOVE and TRANSLATE, one operand is replaced by new data, and the execution of the operation may be affected by the amount of overlap and the manner in which data is fetched or stored. For purposes of evaluating the effect of overlapped...
operands, data is considered to be handled one eight-bit byte at a time. Special rules apply to the operands of MOVE LONG and MOVE INVERSE. See "Interlocks within a Single Instruction" on page 5-91 for how overlap is detected in the access-register mode.

**Binary-Integer Representation**

Binary integers are treated as signed or unsigned.

In an unsigned binary integer, all bits are used to express the absolute value of the number. When two unsigned binary integers of different lengths are added, the shorter number is considered to be extended on the left with zeros.

In some operations, the result is achieved by the use of the one's complement of the number. The one's complement of a number is obtained by inverting each bit of the number, including the sign.

For signed binary integers, the leftmost bit represents the sign, which is followed by the numeric field. Positive numbers are represented in true binary notation with the sign bit set to zero. When the value is zero, all bits are zeros, including the sign bit. Negative numbers are represented in two's-complement binary notation with a one in the sign-bit position.

Specifically, a negative number is represented by the two's complement of the positive number of the same absolute value. The two's complement of a number is obtained by forming the one's complement of the number, adding a value of one in the rightmost bit position, allowing a carry into the sign position, and ignoring any carry out of the sign position.

This number representation can be considered the rightmost portion of an infinitely long representation of the number. When the number is positive, all bits to the left of the most significant bit of the number are zeros. When the number is negative, these bits are ones. Therefore, when a signed operand must be extended with bits on the left, the extension is achieved by setting these bits equal to the sign bit of the operand.

The notation for signed binary integers does not include a negative zero. It has a number range in which, for a given length, the set of negative nonzero numbers is one larger than the set of positive nonzero numbers. The maximum positive number consists of a sign bit of zero followed by all ones, whereas the maximum negative number (the negative number with the greatest absolute value) consists of a sign bit of one followed by all zeros.

A signed binary integer of either sign, except for zero and the maximum negative number, can be changed to a number of the same magnitude but opposite sign by forming its two's complement. Forming the two's complement of a number is equivalent to subtracting the number from zero. The two's complement of zero is zero.

The two's complement of the maximum negative number cannot be represented in the same number of bits. When an operation, such as LOAD COMPLEMENT, attempts to produce the two's complement of the maximum negative number, the result is the maximum negative number, and a fixed-point-overflow exception is recognized. An overflow does not result, however, when the maximum negative number is complemented as an intermediate result but the final result is within the representable range. An example of this case is a subtraction of the maximum negative number from -1. The product of two maximum negative numbers of a given length is representable as a positive number of double that length.

In discussions of signed binary integers in this publication, a signed binary integer includes the sign bit. Thus, the expression “32-bit signed binary integer” denotes an integer with 31 numeric bits and a sign bit, and the expression “64-bit signed binary integer” denotes an integer with 63 numeric bits and a sign bit.

In an arithmetic operation, a carry out of the numeric field of a signed binary integer is carried into the sign bit. However, in algebraic left-shifting, the sign bit does not change even if significant numeric bits are shifted out.

**Programming Notes:**

1. An alternate way of forming the two's complement of a signed binary integer is to invert all bits to the left of the rightmost one bit, leaving the rightmost one bit and all zero bits to the right of it unchanged.
2. The numeric bits of a signed binary integer may be considered to represent a positive value, with the sign representing a value of either zero or the maximum negative number.

**Binary Arithmetic**

Many of the instructions that perform a register-and-storage or register-and-register binary-arithmetic operation are provided in sets of three instructions corresponding to three different combinations of operand lengths. These three instructions have the same name but different operation codes and mnemonics. For example, ADD (A) operates on 32-bit operands and produces a 32-bit result, ADD (AG) operates on 64-bit operands and produces a 64-bit result, and ADD (AGF) operates on a 64-bit operand and a 32-bit operand and produces a 64-bit result. The letter “G” alone in the mnemonic indicates a completely 64-bit operation, and the letters “GF” indicate a 32-to-64-bit operation.

In a 32-to-64-bit operation, the intermediate result is 64 bits. LOAD COMPLEMENT (LCGFR) forms the two's complement of the maximum negative 32-bit number without recognizing overflow.

A 32-bit operand in a general register is in bit positions 32-63 of the register. In an operation on the operand, such as by ADD (A), bits 0-31 of the register are unused and remain unchanged. A 64-bit operand in a general register is in bit positions 0-63 of the register, and all of the bits participate in an operation on the operand, such as by ADD (AG). However, some instructions, which do not have “G” in their mnemonics, use a 64-bit operand of which the leftmost 32 bits are in bit positions 32-63 of the even register of an even-odd general-register pair, and the rightmost 32 bits are in bit positions 32-63 of the odd register of the pair.

The bits of a 32-bit operand in storage are numbered 0-31. When the operand is in bit positions 32-63 of a general register, the bits are numbered 32-63.

**Signed Binary Arithmetic**

**Addition and Subtraction**

Addition of signed binary integers is performed by adding all bits of each operand, including the sign bits. When one of the operands is shorter, the shorter operand is considered to be extended on the left to the length of the longer operand by propagating the sign-bit value.

For a 32-bit signed binary integer in a general register, the sign bit is bit 32 of the register. For a 64-bit signed binary integer in a general register, the sign bit is bit 0 of the register.

Subtraction is performed by adding the one's complement of the second operand and a value of one to the first operand.

**Fixed-Point Overflow**

A fixed-point-overflow condition exists for signed binary addition or subtraction when the carry out of the sign-bit position and the carry out of the leftmost numeric bit position disagree. Detection of an overflow does not affect the result produced by the addition. In mathematical terms, signed addition and subtraction produce a fixed-point overflow when the result is outside the range of representation for signed binary integers. Specifically, for ADD (A) and SUBTRACT (S), which operate on 32-bit signed binary integers, there is an overflow when the proper result would be greater than or equal to $2^{31}$ or less than $-2^{31}$. The actual result placed in the general register after an overflow differs from the proper result by $2^{30}$. A fixed-point overflow causes a program interruption if allowed by the program mask. Similarly, for ADD (AG) and SUBTRACT (SG), which operate on 64-bit signed binary integers, there is an overflow when the proper result would be greater than or equal to $2^{63}$ or less than $-2^{63}$, and the actual result placed in the general register after an overflow differs from the proper result by $2^{62}$. ADD (AGF) and SUBTRACT (SGF) have the same 64-bit result and overflow rules as ADD (AG) and SUBTRACT (SG).

The instructions SHIFT LEFT SINGLE and SHIFT LEFT DOUBLE produce an overflow when the result is outside the range of representation for signed binary integers. The actual result differs from that for addition and subtraction in that the sign of the result remains the same as the original sign.
Unsigned Binary Arithmetic

Addition of unsigned binary integers is performed by adding all bits of each operand. Subtraction is performed by adding the one's complement of the second operand (the subtrahend) and a value of one to the first operand (the subtrahend). In any case, when one of the operands is shorter, the shorter operand is considered to be extended on the left with zeros. During subtraction, this extension applies before an operand is complemented, and it applies to the value of one.

Unsigned binary arithmetic is used in address arithmetic for adding the X, B, and D fields. (See "Address Generation" on page 5-8.) It is also used to obtain the addresses of the function bytes in TRANSLATE and TRANSLATE AND TEST. Furthermore, unsigned binary arithmetic is used on 32-bit or 64-bit unsigned binary integers by ADD LOGICAL, ADD LOGICAL WITH CARRY, DIVIDE LOGICAL, MULTIPLY LOGICAL, SUBTRACT LOGICAL, and SUBTRACT LOGICAL WITH BORROW.

Given the same length operands, ADD (A, AG, AGF) and ADD LOGICAL (AL, ALG, ALGF) produce the same 32-bit or 64-bit result. The instructions differ only in the interpretation of this result. ADD interprets the result as a signed binary integer and inspects it for sign, magnitude, and overflow to set the condition code accordingly. ADD LOGICAL interprets the result as an unsigned binary integer and sets the condition code according to whether the result is zero and whether there was a carry out of bit position 32, for a 32-bit integer, or out of bit position 0 for a 64-bit integer. Such a carry is not considered an overflow, and no program interruption for overflow can occur for ADD LOGICAL.

SUBTRACT LOGICAL differs from ADD LOGICAL in that the one's complement of the second operand and a value of one are added to the first operand.

For ADD LOGICAL WITH CARRY, a carry from a previous operation is represented by a one value of bit 18 of the current PSW. Bit 18 is the leftmost bit of the two-bit condition code in the PSW. For SUBTRACT LOGICAL WITH BORROW, a borrow from a previous operation is represented by a zero value of bit 18. A borrow is equivalent to the absence of a carry.

Programming Notes:

1. Logical addition and subtraction may be used to perform arithmetic on multiple-precision binary-integer operands. Thus, for multiple-precision addition, ADD LOGICAL can be used to add the lowest-order corresponding parts of the operands, and ADD LOGICAL WITH CARRY can be used to add the other corresponding parts of the operands, moving from right to left in the operands. If the multiple-precision operands are signed, ADD should be used on the highest-order parts. The condition code then indicates any overflow or the proper sign and magnitude of the entire result; an overflow is also indicated by a program interruption for fixed-point overflow if allowed by the program mask. When ADD is used, a value of one must be added to the sum of the highest-order parts if the condition code indicated there was a carry from the addition of the next-lower parts.

2. Another use for ADD LOGICAL is to increment values representing binary counters, which are allowed to wrap around from all ones to all zeros without indicating overflow.

Signed and Logical Comparison

Comparison operations determine whether two operands are equal or not and, for most operations, which of two unequal operands is the greater (high). Signed-binary-comparison operations are provided which treat the operands as signed binary integers, and logical-comparison operations are provided which treat the operands as unsigned binary integers or as unstructured data.

COMPARE (C, CG, CGF) and COMPARE HALFWORD are signed-binary-comparison operations. These instructions are equivalent to SUBTRACT (S, SG, SGF) and SUBTRACT HALFWORD without replacing either operand, the resulting difference being used only to set the condition code. The operations permit comparison of numbers of opposite sign which differ by 2^63 or more. Thus, unlike SUBTRACT, COMPARE cannot cause overflow.

Logical comparison of two operands is performed byte by byte, in a left-to-right sequence. The
operands are equal when all their bytes are equal. When the operands are unequal, the comparison result is determined by a left-to-right comparison of corresponding bit positions in the first unequal pair of bytes: the zero bit in the first unequal pair of bits indicates the low operand, and the one bit the high operand. Since the remaining bit and byte positions do not change the comparison, it is not necessary to continue comparing unequal operands beyond the first unequal bit pair.

Instructions

The general instructions and their mnemonics, formats, and operation codes are listed in Figure 7-1 on page 7-9. The figure also indicates which instructions are new in z/Architecture as compared to ESA/390, when the condition code is set, the instruction fields that designate access registers, and the exceptional conditions in operand designations, data, or results that cause a program interruption.

The instructions that are new in z/Architecture are indicated in Figure 7-1 by “N.” A few of the instructions that are new in z/Architecture have also been added to ESA/390, and these are indicated by “N3.”

When the operands of an instruction are 32-bit operands, the mnemonic for the instruction does not include a letter indicating the operand length. If there is an instruction with the same name but with 64-bit operands, its mnemonic includes the letter “G.” If there is an instruction with the same name but with a 64-bit first operand and a 32-bit second operand, its mnemonic includes the letters “GF.”

Instructions

1. Trimodal addressing affects the general instructions only in the manner in which logical storage addresses are handled, except as follows.

The instructions BRANCH AND LINK (BAL, BALR), BRANCH AND SAVE (BAS, BASR), BRANCH AND SAVE AND SET MODE, BRANCH AND SET MODE, BRANCH RELATIVE AND SAVE, and BRANCH RELATIVE AND SAVE LONG place information in bit positions 32-39 of general register R₁ as in ESA/390 in the 24-bit or 31-bit addressing mode or place address bits in those bit positions in the 64-bit addressing mode.

The instruction BRANCH AND SAVE AND SET MODE places a zero in bit position 63 of general register R₁ in the 24-bit or 31-bit addressing mode or places a one in that bit position in the 64-bit addressing mode.

The instruction BRANCH AND SET MODE leaves the contents of bit position 63 of general register R₁ unchanged in the 24-bit or 31-bit addressing mode but places a one in that bit position in the 64-bit addressing mode.

The following instructions leave bits 0-31 of a general register unchanged in the 24-bit or 31-bit addressing mode but place or update address or length information in them in the 64-bit addressing mode. Also, the leftmost byte of the results in registers may be handled differently depending on whether the addressing mode is the 24-bit or the 31-bit mode.

- BRANCH AND LINK (BAL, BALR)
- BRANCH AND SAVE (BAS, BASR)
- BRANCH AND SAVE AND SET MODE
- BRANCH RELATIVE AND SAVE
- BRANCH RELATIVE AND SAVE LONG
- CHECKSUM

A detailed definition of instruction formats, operand designation and length, and address generation is contained in "Instructions" on page 5-2.

Exceptions to the general rules stated in that section are explicitly identified in the individual instruction descriptions.

Note: In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designations for the assembler language are shown with each instruction. For LOAD AND TEST with 32-bit operands, for example, LTR is the mnemonic and R₁,R₂ the operand designation.
• COMPARE AND FORM CODEWORD
• COMPARE LOGICAL LONG
• COMPARE LOGICAL LONG EXTENDED
• COMPARE LOGICAL LONG UNICODE
• COMPARE LOGICAL STRING
• COMPARE UNTIL SUBSTRING EQUAL
• COMPRESSION CALL
• CONVERT UNICODE TO UTF-8
• CONVERT UTF-8 TO UNICODE
• LOAD ADDRESS
• LOAD ADDRESS EXTENDED
• LOAD ADDRESS RELATIVE LONG
• MOVE LONG
• MOVE LONG EXTENDED
• MOVE LONG UNICODE
• MOVE STRING
• SEARCH STRING
• TRANSLATE EXTENDED
• TRANSLATE AND TEST
• TRANSLATE ONE TO ONE
• TRANSLATE ONE TO TWO
• TRANSLATE TWO TO ONE
• TRANSLATE TWO TO TWO
• UPDATE TREE

The instructions in the preceding list are sometimes called modal instructions.

2. Bits 0-31 of general registers are changed by two types of instructions. The first type is a modal instruction, as listed in the preceding note, when the instruction is executed in the 64-bit addressing mode. The second type is an instruction having, independent of the addressing mode, either a 64-bit result operand in a single general register or a 128-bit result operand in an even-odd general-register pair.

Most of the instructions of the second type are indicated by a “G,” either alone or in “GF,” in their mnemonics. The other instructions that change or may change bits 0-31 of a general register regardless of the current addressing mode are:

• AND IMMEDIATE (NIHH, NIHL only)
• INSERT CHARACTERS UNDER MASK (ICMH only)
• INSERT IMMEDIATE (IIHH, IIHL only)
• LOAD LOGICAL IMMEDIATE (LLIHH, LLIHL only)
• LOAD MULTIPLE DISJOINT
• LOAD MULTIPLE HIGH
• LOAD PAIR FROM QUADWORD

• OR IMMEDIATE (OIHH, OIHL only)

All of the instructions of the second type are sometimes referred to as “G-type” instructions.

If a program is not executed in the 64-bit addressing mode and does not contain a G-type instruction, it cannot change bits 0-31 of any general register.

3. It is not intended or expected that old programs not containing G-type instructions will be able to be executed successfully in the 64-bit addressing mode. However, this may be possible, particularly if, by programming convention, bits 0-31 of the general registers are always all zeros when an old program is given control.

4. The following additional general instructions are available when the extended-translation facility 2 is installed:

• COMPARE LOGICAL LONG UNICODE
• MOVE LONG UNICODE
• PACK ASCII
• PACK UNICODE
• TRANSLATE ONE TO ONE
• TRANSLATE ONE TO TWO
• TRANSLATE TWO TO ONE
• TRANSLATE TWO TO TWO
• UNPACK ASCII
• UNPACK UNICODE

5. The long-displacement facility uses new instruction formats, named RSY, RXY, and SIY, to provide 20-bit signed displacements. In connection with the long-displacement facility, all previously existing general instructions of the RSE or RXE format are changed to be of format RSY or RXY, respectively, where the new formats differ from the old by using a previously unused byte, now named DH, in the instructions. When the long-displacement facility is installed, the displacement for an instruction operand address is formed by appending DH on the left of the previous displacement field, now named DL, of the instruction. When the long-displacement facility is not installed, eight zero bits are appended on the left of DL, and DH is ignored.

The following additional general instruction is available when the long-displacement facility is installed:

• LOAD BYTE
The following additional RSY-format versions of general instructions are available when the long-displacement facility is installed.

- COMPARE AND SWAP
- COMPARE DOUBLE AND SWAP
- COMPARE LOGICAL CHARACTERS UNDER MASK
- LOAD ACCESS MULTIPLE
- LOAD MULTIPLE
- STORE ACCESS MULTIPLE
- STORE CHARACTERS UNDER MASK
- STORE MULTIPLE

The following additional RXY-format versions of general instructions are available when the long-displacement facility is installed.

- ADD
- ADD HALFWORD
- ADD LOGICAL
- AND
- COMPARE
- COMPARE HALFWORD
- COMPARE LOGICAL
- CONVERT TO BINARY
- CONVERT TO DECIMAL
- EXCLUSIVE OR
- INSERT CHARACTER
- INSERT CHARACTER UNDER MASK
- LOAD
- LOAD ADDRESS
- LOAD HALFWORD
- MULTIPLY SINGLE
- OR
- STORE
- STORE CHARACTER
- STORE HALFWORD
- SUBTRACT
- SUBTRACT HALFWORD
- SUBTRACT LOGICAL

The following additional SIY-format versions of general instructions are available when the long-displacement facility is installed.

- AND
- COMPARE LOGICAL
- EXCLUSIVE OR
- MOVE
- OR
- TEST UNDER MASK

6. The following additional general instructions are available when the message-security assist is installed:

- CIPHER MESSAGE
- CIPHER MESSAGE WITH CHAINING
- COMPUTE INTERMEDIATE MESSAGE DIGEST
- COMPUTE LAST MESSAGE DIGEST
- COMPUTE MESSAGE AUTHENTICATION CODE

7. The following additional general instructions are available when the extended-translation facility 3 is installed:

- CONVERT UTF-16 TO UTF-32
- CONVERT UTF-32 TO UTF-16
- CONVERT UTF-32 TO UTF-8
- CONVERT UTF-8 TO UTF-32
- SEARCH STRING UNICODE
- TRANSLATE AND TEST REVERSED

Additionally, CONVERT UNICODE TO UTF-8 (CUUTF) and CONVERT UTF-8 TO UNICODE (CU12F) are renamed to CONVERT UTF-16 TO UTF-8 (CU21) and CONVERT UTF-8 TO UTF-16 (CU12), respectively, in order to conform to the names used by this facility. The old names continue to be recognized.
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (32)</td>
<td>AR</td>
<td>RR C</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (64)</td>
<td>AGR</td>
<td>RRE C N</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (64&lt;32)</td>
<td>AGFR</td>
<td>RRE C N</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>A</td>
<td>RX C A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>AY</td>
<td>RXY C LD A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (64)</td>
<td>AG</td>
<td>RXY C N A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (64&lt;32)</td>
<td>AGF</td>
<td>RXY C N A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>A</td>
<td>RX C A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>AHI</td>
<td>RI C</td>
<td>IF</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>A</td>
<td>RXY C LD A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>AL</td>
<td>RR C</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (64)</td>
<td>ALGR</td>
<td>RRE C N</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (64&lt;32)</td>
<td>ALGFR</td>
<td>RRE C N</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>AL</td>
<td>RX C A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>ALY</td>
<td>RXY C LD A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>A</td>
<td>RX C A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>A</td>
<td>RX C A</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>AHI</td>
<td>RI C</td>
<td>IF</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>A</td>
<td>RXY C LD A</td>
<td>IF</td>
</tr>
<tr>
<td>AND (32)</td>
<td>NR</td>
<td>RR C</td>
<td>IF</td>
</tr>
<tr>
<td>AND (32)</td>
<td>N</td>
<td>RX C A</td>
<td>IF</td>
</tr>
<tr>
<td>AND (32)</td>
<td>NY</td>
<td>RXY C LD A</td>
<td>IF</td>
</tr>
<tr>
<td>AND (character)</td>
<td>NC</td>
<td>SS C A</td>
<td>IF</td>
</tr>
<tr>
<td>AND (immediate)</td>
<td>NI</td>
<td>SI C A</td>
<td>IF</td>
</tr>
<tr>
<td>AND (immediate)</td>
<td>NIY</td>
<td>SIY C LD A</td>
<td>IF</td>
</tr>
<tr>
<td>AND IMMEDIATE (high high)</td>
<td>NIHH</td>
<td>RI C N</td>
<td>IF</td>
</tr>
<tr>
<td>AND IMMEDIATE (high low)</td>
<td>NIHL</td>
<td>RI C N</td>
<td>IF</td>
</tr>
<tr>
<td>AND IMMEDIATE (low high)</td>
<td>NILH</td>
<td>RI C N</td>
<td>IF</td>
</tr>
<tr>
<td>AND IMMEDIATE (low low)</td>
<td>NILL</td>
<td>RI C N</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH AND LINK</td>
<td>BALR</td>
<td>RR T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH AND SAVE</td>
<td>BASR</td>
<td>RR T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH AND SAVE</td>
<td>BAS</td>
<td>RR T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH AND SAVE AND SET MODE</td>
<td>BASSM</td>
<td>RR T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH AND SET MODE</td>
<td>BSM</td>
<td>RR T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH ON CONDITION</td>
<td>BCR</td>
<td>RR T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH ON CONDITION</td>
<td>BC</td>
<td>RX T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH ON COUNT (32)</td>
<td>BCTR</td>
<td>RR T B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH ON COUNT (64)</td>
<td>BCTGR</td>
<td>RRE N B</td>
<td>IF</td>
</tr>
<tr>
<td>BRANCH ON COUNT (32)</td>
<td>BCT</td>
<td>RX T B</td>
<td>IF</td>
</tr>
</tbody>
</table>

Figure 7-1 (Part 1 of 9). Summary of General Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH ON COUNT (64)</td>
<td>BCTG</td>
<td>RXY N</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH ON INDEX HIGH (32)</td>
<td>BXH</td>
<td>RS Y N</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH ON INDEX HIGH (64)</td>
<td>BXHG</td>
<td>RSY N</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH ON INDEX LOW OR EQUAL (32)</td>
<td>BXLE</td>
<td>RSY N</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH ON INDEX LOW OR EQUAL (64)</td>
<td>BXLE</td>
<td>RSY N</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH RELATIVE AND SAVE</td>
<td>BRAS</td>
<td>RI N</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH RELATIVE AND SAVE LONG</td>
<td>BRASL</td>
<td>RIL N3</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON CONDITION</td>
<td>BRC</td>
<td>RIL N3</td>
<td>B</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON COUNT (32)</td>
<td>BRCTG</td>
<td>RRE C N</td>
<td>A</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON COUNT (64)</td>
<td>BRCTG</td>
<td>RRE C N</td>
<td>A</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON INDEX HIGH (32)</td>
<td>BRXH</td>
<td>RY C LD</td>
<td>A</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON INDEX HIGH (64)</td>
<td>BRXH</td>
<td>RY C LD</td>
<td>A</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON INDEX LOW OR EQUAL (32)</td>
<td>BRXLEG</td>
<td>RY C N</td>
<td>A</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON INDEX LOW OR EQUAL (64)</td>
<td>BRXLEG</td>
<td>RY C N</td>
<td>A</td>
</tr>
<tr>
<td>CHECKSUM</td>
<td>CKSM</td>
<td>RRE C</td>
<td>A SP</td>
</tr>
<tr>
<td>CIPHER MESSAGE</td>
<td>KM</td>
<td>RRE C</td>
<td>A SP</td>
</tr>
<tr>
<td>CIPHER MESSAGE WITH CHAINING</td>
<td>KMC</td>
<td>RRE C</td>
<td>A SP</td>
</tr>
<tr>
<td>COMPARE (32)</td>
<td>CR</td>
<td>RX C</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE (64)</td>
<td>CGR</td>
<td>RX C N</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE (64&lt;32)</td>
<td>CR</td>
<td>RX C N</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE AND FORM CODEWORD</td>
<td>CFC</td>
<td>S C</td>
<td>A SP</td>
</tr>
<tr>
<td>COMPARE AND SWAP (32)</td>
<td>CS</td>
<td>RS C</td>
<td>A SP</td>
</tr>
<tr>
<td>COMPARE AND SWAP (32)</td>
<td>CSY</td>
<td>RSY C LD</td>
<td>A SP</td>
</tr>
<tr>
<td>COMPARE AND SWAP (64)</td>
<td>CGS</td>
<td>RSY C N</td>
<td>A SP</td>
</tr>
<tr>
<td>COMPARE DOUBLE AND SWAP (32)</td>
<td>CDS</td>
<td>RS C</td>
<td>A SP</td>
</tr>
<tr>
<td>COMPARE DOUBLE AND SWAP (32)</td>
<td>CDSY</td>
<td>RSY C LD</td>
<td>A SP</td>
</tr>
<tr>
<td>COMPARE HALFWORD</td>
<td>CH</td>
<td>RX C</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE HALFWORD</td>
<td>CHY</td>
<td>RX C LD</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE HALFWORD IMMEDIATE (32)</td>
<td>CHI</td>
<td>RI C</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE HALFWORD IMMEDIATE (64)</td>
<td>CHI</td>
<td>RI C N</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (32)</td>
<td>CLR</td>
<td>RR C</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (32)</td>
<td>CLY</td>
<td>RX Y C LD</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (64)</td>
<td>CLGR</td>
<td>RRE C N</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>CLGR</td>
<td>RRE C N</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (32)</td>
<td>CL</td>
<td>RX C</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (64)</td>
<td>CLG</td>
<td>RX Y C N</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>CLG</td>
<td>RX Y C N</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (character)</td>
<td>CLC</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE LOGICAL (immediate)</td>
<td>CLI</td>
<td>SI C</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure 7-1 (Part 2 of 9). Summary of General Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARE LOGICAL (Immediate)</td>
<td>CLMY</td>
<td>RS C LD</td>
<td>B1</td>
</tr>
<tr>
<td>COMPARE LOGICAL C. UNDER MASK (High)</td>
<td>CLMH</td>
<td>RSY C N</td>
<td>B2</td>
</tr>
<tr>
<td>COMPARE LOGICAL C. UNDER MASK (Low)</td>
<td>CLM</td>
<td>RS C</td>
<td>B2</td>
</tr>
<tr>
<td>COMPARE LOGICAL C. UNDER MASK (Low)</td>
<td>CLMY</td>
<td>RSY C LD</td>
<td>B2</td>
</tr>
<tr>
<td>COMPARE LOGICAL LONG</td>
<td>CLCL</td>
<td>RR C</td>
<td>R1 R2</td>
</tr>
<tr>
<td>COMPARE LOGICAL LONG EXTENDED</td>
<td>CLCLE</td>
<td>RS C</td>
<td>R1 R3</td>
</tr>
<tr>
<td>COMPARE LOGICAL STRING</td>
<td>CLST</td>
<td>RRE C</td>
<td>R1 R2</td>
</tr>
<tr>
<td>COMPARE UNTIL SUBSTRING EQUAL</td>
<td>CUSE</td>
<td>RRE C</td>
<td>R1 R2</td>
</tr>
<tr>
<td>COMPUTE INTERMEDIATE MESSAGE DIGEST</td>
<td>KIMD</td>
<td>RRE C MS</td>
<td>ST R2</td>
</tr>
<tr>
<td>COMPUTE LAST MESSAGE DIGEST</td>
<td>KLMID</td>
<td>RRE C MS</td>
<td>ST R2</td>
</tr>
<tr>
<td>COMPUTE MESSAGE AUTHENTICATION CODE</td>
<td>KMAC</td>
<td>RRE C MS</td>
<td>ST R2</td>
</tr>
<tr>
<td>CONVERT TO BINARY (32)</td>
<td>CVB</td>
<td>RX</td>
<td>ST B2</td>
</tr>
<tr>
<td>CONVERT TO BINARY (32)</td>
<td>CVBY</td>
<td>RXY LD</td>
<td>ST B2</td>
</tr>
<tr>
<td>CONVERT TO BINARY (64)</td>
<td>CVBG</td>
<td>RXY N</td>
<td>ST B2</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (32)</td>
<td>CVD</td>
<td>RX</td>
<td>ST B2</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (32)</td>
<td>CVDY</td>
<td>RXY LD</td>
<td>ST B2</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (64)</td>
<td>CVDG</td>
<td>RX N</td>
<td>ST B2</td>
</tr>
<tr>
<td>CONVERT UNICOD TO UTF-8</td>
<td>CUUTF</td>
<td>RRE C</td>
<td>ST R1 R2</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-32</td>
<td>CU24</td>
<td>RRE C E3</td>
<td>ST R1 R2</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-8</td>
<td>CU21</td>
<td>RRE C E3</td>
<td>ST R1 R2</td>
</tr>
<tr>
<td>CONVERT UTF-32 TO UTF-16</td>
<td>CU42</td>
<td>RRE C E3</td>
<td>ST R1 R2</td>
</tr>
<tr>
<td>CONVERT UTF-32 TO UTF-8</td>
<td>CU41</td>
<td>RRE C E3</td>
<td>ST R1 R2</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UNICODE</td>
<td>CUTFU</td>
<td>RRE C</td>
<td>ST R1 R2</td>
</tr>
<tr>
<td>DIVIDE (32&lt;64)</td>
<td>DRL</td>
<td>RR SP</td>
<td>DK B2</td>
</tr>
<tr>
<td>DIVIDE (32&lt;64)</td>
<td>DSR</td>
<td>RR SP</td>
<td>DK B2</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (32&lt;64)</td>
<td>DLR</td>
<td>RR N3</td>
<td>B997</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>DLGR</td>
<td>RR N</td>
<td>B987</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (32&lt;64)</td>
<td>DL</td>
<td>RXY N3</td>
<td>B397</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>DLG</td>
<td>RXY N</td>
<td>B387</td>
</tr>
<tr>
<td>DIVIDE SINGLE (64)</td>
<td>DSGR</td>
<td>RR N</td>
<td>B900</td>
</tr>
<tr>
<td>DIVIDE SINGLE (64&lt;32)</td>
<td>DSGFR</td>
<td>RRE N</td>
<td>B91D</td>
</tr>
<tr>
<td>DIVIDE SINGLE (64)</td>
<td>DSG</td>
<td>RXY N</td>
<td>B30D</td>
</tr>
<tr>
<td>EXCLUSIVE OR (32)</td>
<td>XR</td>
<td>RR C</td>
<td>17</td>
</tr>
<tr>
<td>EXCLUSIVE OR (32)</td>
<td>XGR</td>
<td>RRE C N</td>
<td>B982</td>
</tr>
<tr>
<td>EXCLUSIVE OR (32)</td>
<td>X</td>
<td>RX C</td>
<td>B2 57</td>
</tr>
<tr>
<td>EXCLUSIVE OR (32)</td>
<td>XY</td>
<td>RXY C LD</td>
<td>B2 357</td>
</tr>
<tr>
<td>EXCLUSIVE OR (64)</td>
<td>XG</td>
<td>RXY C N</td>
<td>B2 382</td>
</tr>
<tr>
<td>EXCLUSIVE OR (character)</td>
<td>XC</td>
<td>SS C</td>
<td>ST B2 B7</td>
</tr>
<tr>
<td>EXCLUSIVE OR (immediate)</td>
<td>XI</td>
<td>SI C</td>
<td>ST B1 97</td>
</tr>
</tbody>
</table>

Figure 7-1 (Part 3 of 9). Summary of General Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCLUSIVE OR (immediate)</td>
<td>XIY</td>
<td>SIY C LD A</td>
<td>ST</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>EX</td>
<td>RX AI SP EX</td>
<td></td>
</tr>
<tr>
<td>EXTRACT ACCESS</td>
<td>EAR</td>
<td>RRE N3</td>
<td></td>
</tr>
<tr>
<td>EXTRACT PSW</td>
<td>EPSW</td>
<td>RX A</td>
<td></td>
</tr>
<tr>
<td>INSERT CHARACTER</td>
<td>IC</td>
<td>RX</td>
<td></td>
</tr>
<tr>
<td>INSERT CHARACTER UNDER MASK (high)</td>
<td>ICY</td>
<td>RXY LD A</td>
<td></td>
</tr>
<tr>
<td>INSERT CHARACTER UNDER MASK (low)</td>
<td>ICM</td>
<td>RS C N A</td>
<td></td>
</tr>
<tr>
<td>INSERT CHARACTER UNDER MASK (low)</td>
<td>ICMY</td>
<td>RSY C LD A</td>
<td></td>
</tr>
<tr>
<td>INSERT IMMEDIATE (high high)</td>
<td>IIHH</td>
<td>RRE N3</td>
<td></td>
</tr>
<tr>
<td>INSERT IMMEDIATE (high low)</td>
<td>IIHL</td>
<td>RRE N A SP</td>
<td></td>
</tr>
<tr>
<td>INSERT IMMEDIATE (low high)</td>
<td>IILH</td>
<td>RRE N A</td>
<td></td>
</tr>
<tr>
<td>INSERT IMMEDIATE (low low)</td>
<td>IILL</td>
<td>RRE N A</td>
<td></td>
</tr>
<tr>
<td>INSERT PROGRAM MASK</td>
<td>IPM</td>
<td>RRE</td>
<td></td>
</tr>
<tr>
<td>LOAD (32)</td>
<td>LR</td>
<td>RX A</td>
<td></td>
</tr>
<tr>
<td>LOAD (64)</td>
<td>LGR</td>
<td>RRE N</td>
<td></td>
</tr>
<tr>
<td>LOAD (64&lt;32)</td>
<td>LGFR</td>
<td>RRE N</td>
<td></td>
</tr>
<tr>
<td>LOAD ADDRESS EXTENDED</td>
<td>LAE</td>
<td>RX N3</td>
<td>U1 BP</td>
</tr>
<tr>
<td>LOAD ADDRESS RELATIVE LONG</td>
<td>LARL</td>
<td>RIL N3</td>
<td></td>
</tr>
<tr>
<td>LOAD AND TEST (32)</td>
<td>LTR</td>
<td>RR C</td>
<td></td>
</tr>
<tr>
<td>LOAD AND TEST (64)</td>
<td>LTGR</td>
<td>RRE C N</td>
<td></td>
</tr>
<tr>
<td>LOAD AND TEST (64&lt;32)</td>
<td>LTGFR</td>
<td>RRE C N</td>
<td></td>
</tr>
<tr>
<td>LOAD BYTE (32)</td>
<td>LB</td>
<td>RX LD A</td>
<td></td>
</tr>
<tr>
<td>LOAD BYTE (64)</td>
<td>LGB</td>
<td>RX LD A</td>
<td></td>
</tr>
<tr>
<td>LOAD COMPLEMENT (32)</td>
<td>LCR</td>
<td>RR C</td>
<td></td>
</tr>
<tr>
<td>LOAD COMPLEMENT (64)</td>
<td>LCR</td>
<td>RRE C N</td>
<td></td>
</tr>
<tr>
<td>LOAD COMPLEMENT (64&lt;32)</td>
<td>LCGFR</td>
<td>RRE C N</td>
<td></td>
</tr>
<tr>
<td>LOAD HALFWORD (32)</td>
<td>LH</td>
<td>RX A</td>
<td></td>
</tr>
<tr>
<td>LOAD HALFWORD (32)</td>
<td>LHY</td>
<td>RXY LD A</td>
<td></td>
</tr>
<tr>
<td>LOAD HALFWORD (64)</td>
<td>LGH</td>
<td>RXY N A</td>
<td></td>
</tr>
<tr>
<td>LOAD HALFWORD IMMEDIATE (32)</td>
<td>LHI</td>
<td>RI A</td>
<td></td>
</tr>
<tr>
<td>LOAD HALFWORD IMMEDIATE (64)</td>
<td>LGHI</td>
<td>RI N</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL (64&lt;32)</td>
<td>LLGFR</td>
<td>RRE N</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL (64&lt;32)</td>
<td>LLGF</td>
<td>RXY N A</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL CHARACTER</td>
<td>LLGC</td>
<td>RXY N A</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL HALFWORD</td>
<td>LLGH</td>
<td>RXY N A</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE (high high)</td>
<td>LLIHH</td>
<td>RI N</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-1 (Part 4 of 9). Summary of General Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD LOGICAL IMMEDIATE (high low)</td>
<td>LLIHL</td>
<td>RI N</td>
<td>A5D</td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE (low high)</td>
<td>LLIHL</td>
<td>RI N</td>
<td>A5E</td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE (low low)</td>
<td>LLILL</td>
<td>RI N</td>
<td>A5F</td>
</tr>
<tr>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>LLGTR</td>
<td>RRE N</td>
<td>B917</td>
</tr>
<tr>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>LLGT</td>
<td>RXY N A</td>
<td>E317</td>
</tr>
<tr>
<td>LOAD MULTIPLE (32)</td>
<td>LM</td>
<td>RS A</td>
<td>B2 98</td>
</tr>
<tr>
<td>LOAD MULTIPLE (32)</td>
<td>LMY</td>
<td>RSY LD A</td>
<td>B2 EB98</td>
</tr>
<tr>
<td>LOAD MULTIPLE (64)</td>
<td>LMG</td>
<td>RSY N A</td>
<td>B2 EB04</td>
</tr>
<tr>
<td>LOAD MULTIPLE DISJOINT</td>
<td>LMD</td>
<td>SS N A</td>
<td>B2 EF</td>
</tr>
<tr>
<td>LOAD MULTIPLE HIGH</td>
<td>LMH</td>
<td>RSY N A</td>
<td>B2 EB96</td>
</tr>
<tr>
<td>LOAD NEGATIVE (32)</td>
<td>LNR</td>
<td>RR C</td>
<td>11</td>
</tr>
<tr>
<td>LOAD NEGATIVE (64)</td>
<td>LNGR</td>
<td>RRE C N</td>
<td>B901</td>
</tr>
<tr>
<td>LOAD NEGATIVE (64&lt;32)</td>
<td>LNGFR</td>
<td>RRE C N</td>
<td>B911</td>
</tr>
<tr>
<td>LOAD PAIR FROM QUADWORD</td>
<td>LPQ</td>
<td>RXY N A SP IF</td>
<td>B2 10 E38F</td>
</tr>
<tr>
<td>LOAD PO SITIVE (32)</td>
<td>LPGR</td>
<td>RRE C N IF</td>
<td>B900</td>
</tr>
<tr>
<td>LOAD PO SITIVE (64)</td>
<td>LPGR</td>
<td>RRE C N IF</td>
<td>B910</td>
</tr>
<tr>
<td>LOAD REVERSED (32)</td>
<td>LRV</td>
<td>RXY N A</td>
<td>B2 E31E</td>
</tr>
<tr>
<td>LOAD REVERSED (64)</td>
<td>LRVG</td>
<td>RXY N A</td>
<td>B2 E30F</td>
</tr>
<tr>
<td>LOAD REVERSED (64)&lt;32</td>
<td>LRVG</td>
<td>RXY N A</td>
<td>B900</td>
</tr>
<tr>
<td>LOAD REVERSED (64)</td>
<td>LRVG</td>
<td>RXY N A</td>
<td>B910</td>
</tr>
<tr>
<td>LOAD REVERSED (16)</td>
<td>LRVH</td>
<td>RXY N A</td>
<td>B2 E31F</td>
</tr>
<tr>
<td>LOAD REVERSED (32)</td>
<td>LRVR</td>
<td>RXY N A MO SP</td>
<td>B2 E31F</td>
</tr>
<tr>
<td>LOAD REVERSED (64)</td>
<td>LRVG</td>
<td>RXY N A SP MO</td>
<td>B2 E31F</td>
</tr>
<tr>
<td>MONITOR CALL</td>
<td>MC</td>
<td>SI SP MO</td>
<td>ST B1</td>
</tr>
<tr>
<td>MOVE (character)</td>
<td>MVC</td>
<td>SS A</td>
<td>ST B2</td>
</tr>
<tr>
<td>MOVE (immediate)</td>
<td>MVI</td>
<td>SI A</td>
<td>ST B2</td>
</tr>
<tr>
<td>MOVE (immediate)</td>
<td>MVIY</td>
<td>SIY LD A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MOVE INVERSE</td>
<td>MVCIN</td>
<td>SS A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MOVE LONG</td>
<td>MCVL</td>
<td>RY N A</td>
<td>ST R1</td>
</tr>
<tr>
<td>MOVE LONG EXTENDED</td>
<td>MCVLE</td>
<td>RY N A SP II</td>
<td>ST R1</td>
</tr>
<tr>
<td>MOVE LONG UNICODE</td>
<td>MCVLUC</td>
<td>RSY C E2 A SP</td>
<td>ST R1</td>
</tr>
<tr>
<td>MOVE NUMERICANS</td>
<td>MVN</td>
<td>SS A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MOVE STRING</td>
<td>MVST</td>
<td>RRE C A SP G0</td>
<td>ST R2</td>
</tr>
<tr>
<td>MOVE WITH OFFSET</td>
<td>MVO</td>
<td>SS A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MOVE ZONES</td>
<td>MVZ</td>
<td>SS A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY (64&lt;32)</td>
<td>MR</td>
<td>RR SP</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY (64&lt;32)</td>
<td>M</td>
<td>RX A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD (32)</td>
<td>MH</td>
<td>RX A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD IMMEDIATE (32)</td>
<td>MHI</td>
<td>RI A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD IMMEDIATE (64)</td>
<td>MGHI</td>
<td>RI N</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>MLR</td>
<td>RRE N3 SP</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>MLGR</td>
<td>RRE N SP</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>ML</td>
<td>RXY N A SP</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>ML</td>
<td>RXY N A SP</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (32)</td>
<td>MSR</td>
<td>RRE A</td>
<td>ST B1</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (64)</td>
<td>MSGR</td>
<td>RRE N</td>
<td>ST B1</td>
</tr>
</tbody>
</table>

Figure 7-1 (Part 5 of 9). Summary of General Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>MSGFR</td>
<td>RRE N</td>
<td>B91C</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (32)</td>
<td>MS</td>
<td>RX A</td>
<td>B2 71</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (32)</td>
<td>MSY</td>
<td>RXY LD A</td>
<td>B2 E351</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (64)</td>
<td>MSG</td>
<td>RXY N A</td>
<td>B2 E30C</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>MSGF</td>
<td>RXY N A</td>
<td>B2 E31C</td>
</tr>
<tr>
<td>OR (32)</td>
<td>OR</td>
<td>RR C RRE C N</td>
<td>16</td>
</tr>
<tr>
<td>OR (64)</td>
<td>ORG</td>
<td>RX A B/basesupertwo</td>
<td>56</td>
</tr>
<tr>
<td>OR (32)</td>
<td>ORY</td>
<td>RXY N B/basesupertwo</td>
<td>63</td>
</tr>
<tr>
<td>OR (64)</td>
<td>ORG</td>
<td>RXY N B/basesupertwo</td>
<td>83</td>
</tr>
<tr>
<td>OR (character)</td>
<td>OC</td>
<td>SS C A</td>
<td>ST B1 B2 D6</td>
</tr>
<tr>
<td>OR (immediate)</td>
<td>OI</td>
<td>SS E2 A SP</td>
<td>ST B1 B2 E9</td>
</tr>
<tr>
<td>OR (immediate)</td>
<td>OIY</td>
<td>SS E2 A SP</td>
<td>ST B1 B2 E1</td>
</tr>
<tr>
<td>OR IMMEDIATE (high high)</td>
<td>OILH</td>
<td>RY C A</td>
<td>Rz B25E</td>
</tr>
<tr>
<td>OR IMMEDIATE (high low)</td>
<td>OILL</td>
<td>RY C A</td>
<td>Rz B9BE</td>
</tr>
<tr>
<td>PACK</td>
<td>PACK</td>
<td>SS A</td>
<td>ST B1 B2 F2</td>
</tr>
<tr>
<td>PACK ASCII</td>
<td>PKA</td>
<td>SS E2 A SP</td>
<td>ST B1 B2 E9</td>
</tr>
<tr>
<td>PACK UNICODE</td>
<td>PKU</td>
<td>SS E2 A SP</td>
<td>ST B1 B2 E1</td>
</tr>
<tr>
<td>PERFORM LOCKED OPERATION</td>
<td>PLO</td>
<td>SS C A SP</td>
<td>ST FC EE</td>
</tr>
<tr>
<td>ROTATE LEFT SINGLE LOGICAL (32)</td>
<td>RLL</td>
<td>RSY N3</td>
<td>EB1D</td>
</tr>
<tr>
<td>ROTATE LEFT SINGLE LOGICAL (64)</td>
<td>RLLG</td>
<td>RSY N A</td>
<td>EB1C</td>
</tr>
<tr>
<td>SEARCH STRING</td>
<td>SRST</td>
<td>RRE C A SP</td>
<td>Rz B25E</td>
</tr>
<tr>
<td>SEARCH STRING UNICODE</td>
<td>SRSTU</td>
<td>RRE C E3 A SP</td>
<td>Rz B9BE</td>
</tr>
<tr>
<td>SET ACCESS</td>
<td>SAR</td>
<td>RRE N E N3 SP T</td>
<td>U1 B24E</td>
</tr>
<tr>
<td>SET ADDRESSING MODE (24)</td>
<td>SAM24</td>
<td>E N3 SP T</td>
<td>010C</td>
</tr>
<tr>
<td>SET ADDRESSING MODE (31)</td>
<td>SAM31</td>
<td>E N3 SP T</td>
<td>010D</td>
</tr>
<tr>
<td>SET ADDRESSING MODE (64)</td>
<td>SAM64</td>
<td>E N T</td>
<td>010E</td>
</tr>
<tr>
<td>SET PROGRAM MASK</td>
<td>SPM</td>
<td>RR L</td>
<td>04</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE</td>
<td>SLDA</td>
<td>RS C SP</td>
<td>IF 8F</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE LOGICAL</td>
<td>SLDL</td>
<td>RS SP</td>
<td>IF 8D</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE (32)</td>
<td>SLA</td>
<td>RS C IF</td>
<td>BB 8B</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE (64)</td>
<td>SLAG</td>
<td>RSY C N</td>
<td>EB0B</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL (32)</td>
<td>SLL</td>
<td>RS IF</td>
<td>89</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL (64)</td>
<td>SLLG</td>
<td>RSY N IF</td>
<td>EB0D</td>
</tr>
<tr>
<td>SHIFT RIGHT DOUBLE</td>
<td>SRDA</td>
<td>RS C SP</td>
<td>EB8</td>
</tr>
<tr>
<td>SHIFT RIGHT DOUBLE LOGICAL</td>
<td>SRLD</td>
<td>RS SP</td>
<td>EB8C</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE (32)</td>
<td>SRA</td>
<td>RS C IF</td>
<td>EB0A</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE (64)</td>
<td>SRRG</td>
<td>RSY C N</td>
<td>EB0A</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE LOGICAL (32)</td>
<td>SRL</td>
<td>RS N</td>
<td>BB EB0C</td>
</tr>
<tr>
<td>STORE (32)</td>
<td>ST</td>
<td>RX A</td>
<td>ST B2 50</td>
</tr>
<tr>
<td>STORE (32)</td>
<td>STY</td>
<td>RXY LD A</td>
<td>ST B2 E350</td>
</tr>
<tr>
<td>STORE (64)</td>
<td>STG</td>
<td>RXY N A</td>
<td>ST B2 E324</td>
</tr>
</tbody>
</table>

Figure 7-1 (Part 6 of 9). Summary of General Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mneemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE ACCESS MULTIPLE</td>
<td>STAM</td>
<td>RS A SP</td>
<td>UTB 9B</td>
</tr>
<tr>
<td>STORE ACCESS MULTIPLE</td>
<td>STAMY</td>
<td>RSY LD A SP</td>
<td>UTB EB9B</td>
</tr>
<tr>
<td>STORE CHARACTER</td>
<td>STC</td>
<td>RX A</td>
<td>DB 42</td>
</tr>
<tr>
<td>STORE CHARACTER</td>
<td>STCY</td>
<td>RXY LD A</td>
<td>DB E372</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (high)</td>
<td>STCMH</td>
<td>RSY N A</td>
<td>DB EB2C</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>STCM</td>
<td>RS A</td>
<td>DB BE</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>STCMY</td>
<td>RSY LD A $</td>
<td>DB B2D</td>
</tr>
<tr>
<td>STORE CLOCK</td>
<td>STCK</td>
<td>S C A $</td>
<td>DB B2E</td>
</tr>
<tr>
<td>STORE CLOCK EXTENDED</td>
<td>STCKE</td>
<td>S C A $</td>
<td>DB B278</td>
</tr>
<tr>
<td>STORE HALFWORD</td>
<td>STHY</td>
<td>RX A</td>
<td>DB 40</td>
</tr>
<tr>
<td>STORE HALFWORD</td>
<td>STMY</td>
<td>RSY LD A</td>
<td>DB E90</td>
</tr>
<tr>
<td>STORE MULTIPLE (32)</td>
<td>STM</td>
<td>RSY N A</td>
<td>DB EB24</td>
</tr>
<tr>
<td>STORE MULTIPLE (32)</td>
<td>STMH</td>
<td>RSY N A</td>
<td>DB EB26</td>
</tr>
<tr>
<td>STORE PAIR TO QUADWORD</td>
<td>STPQ</td>
<td>RXY N A SP</td>
<td>DB E370</td>
</tr>
<tr>
<td>STORE REVERSED (16)</td>
<td>STRV</td>
<td>RXY N3 A</td>
<td>DB E33F</td>
</tr>
<tr>
<td>STORE REVERSED (32)</td>
<td>STRV</td>
<td>RXY N3 A</td>
<td>DB E33E</td>
</tr>
<tr>
<td>STORE REVERSED (64)</td>
<td>STRVG</td>
<td>RXY N A</td>
<td>DB E32F</td>
</tr>
<tr>
<td>SUBTRACT (32)</td>
<td>SR</td>
<td>RX C IF</td>
<td>DB 1B</td>
</tr>
<tr>
<td>SUBTRACT (64)</td>
<td>SGR</td>
<td>RRE C N IF</td>
<td>DB B909</td>
</tr>
<tr>
<td>SUBTRACT (64&lt;32)</td>
<td>SGRF</td>
<td>RRE C N IF</td>
<td>DB B919</td>
</tr>
<tr>
<td>SUBTRACT (32)</td>
<td>S</td>
<td>RX C A IF</td>
<td>DB 5B</td>
</tr>
<tr>
<td>SUBTRACT (32)</td>
<td>SY</td>
<td>RXY C LD A IF</td>
<td>DB E35B</td>
</tr>
<tr>
<td>SUBTRACT (64)</td>
<td>SG</td>
<td>RXY C N A IF</td>
<td>DB E309</td>
</tr>
<tr>
<td>SUBTRACT (64&lt;32)</td>
<td>SGF</td>
<td>RXY C N A IF</td>
<td>DB E319</td>
</tr>
<tr>
<td>SUBTRACT HALFWORD</td>
<td>SH</td>
<td>RX C A IF</td>
<td>DB 4B</td>
</tr>
<tr>
<td>SUBTRACT HALFWORD</td>
<td>SHY</td>
<td>RXY C LD A IF</td>
<td>DB E37B</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (32)</td>
<td>SLR</td>
<td>RR C IF</td>
<td>DB 1F</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64)</td>
<td>SLGR</td>
<td>RRE C N IF</td>
<td>DB B90B</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>SLGFR</td>
<td>RRE C N IF</td>
<td>DB B91B</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (32)</td>
<td>SL</td>
<td>RX C A IF</td>
<td>DB 5F</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (32)</td>
<td>SLY</td>
<td>RXY C LD A IF</td>
<td>DB E35F</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64)</td>
<td>SLG</td>
<td>RXY C N A IF</td>
<td>DB E30B</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>SLGF</td>
<td>RXY C N A IF</td>
<td>DB E31B</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (32)</td>
<td>SLBR</td>
<td>RRE C N3 A</td>
<td>DB B999</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (64)</td>
<td>SLBGR</td>
<td>RRE C N A</td>
<td>DB B989</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (32)</td>
<td>SLB</td>
<td>RXY C N3 A</td>
<td>DB E399</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (64)</td>
<td>SLBG</td>
<td>RXY C N A</td>
<td>DB E389</td>
</tr>
<tr>
<td>SUPERVISOR CALL</td>
<td>SVC</td>
<td>I I</td>
<td>DB 0A</td>
</tr>
<tr>
<td>TEST ADDRESSING MODE</td>
<td>TAM</td>
<td>E C N3 A $</td>
<td>DB 010B</td>
</tr>
<tr>
<td>TEST AND SET</td>
<td>TS</td>
<td>S C A $</td>
<td>DB 93</td>
</tr>
<tr>
<td>TEST UNDER MASK</td>
<td>TM</td>
<td>SI C A</td>
<td>DB 91</td>
</tr>
<tr>
<td>TEST UNDER MASK</td>
<td>TMY</td>
<td>SIY C LD A</td>
<td>DB EB51</td>
</tr>
<tr>
<td>TEST UNDER MASK (high high)</td>
<td>TMHH</td>
<td>RI C N</td>
<td>DB A72</td>
</tr>
</tbody>
</table>

*Figure 7-1 (Part 7 of 9). Summary of General Instructions*
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST UNDER MASK (high low)</td>
<td>TMHL</td>
<td>RI C N</td>
<td>A73</td>
</tr>
<tr>
<td>TEST UNDER MASK (low high)</td>
<td>TMLH</td>
<td>RI C N</td>
<td>A70</td>
</tr>
<tr>
<td>TEST UNDER MASK (low low)</td>
<td>TMLL</td>
<td>RI C N</td>
<td>A71</td>
</tr>
<tr>
<td>TEST UNDER MASK HIGH</td>
<td>TMH</td>
<td>RI C</td>
<td>A70</td>
</tr>
<tr>
<td>TEST UNDER MASK LOW</td>
<td>TML</td>
<td>RI C</td>
<td>A71</td>
</tr>
<tr>
<td>TRANSLATE</td>
<td>TR</td>
<td>SS A</td>
<td>ST B1 B2 DC</td>
</tr>
<tr>
<td>TRANSLATE AND TEST</td>
<td>TRT</td>
<td>SS C A</td>
<td>GM B1 B2 DD</td>
</tr>
<tr>
<td>TRANSLATE AND TEST REVERSED</td>
<td>TRTR</td>
<td>SS C E3 A</td>
<td>GM B1 B2 D0</td>
</tr>
<tr>
<td>TRANSLATE EXTENDED</td>
<td>TRE</td>
<td>RRE C A SP</td>
<td>ST R1 R2 B2A5</td>
</tr>
<tr>
<td>TRANSLATE ONE TO ONE</td>
<td>TROO</td>
<td>RRE C E2 A SP</td>
<td>ST RM R2 B993</td>
</tr>
<tr>
<td>TRANSLATE ONE TO TWO</td>
<td>TROT</td>
<td>RRE C E2 A SP</td>
<td>GM ST R1 R2 B992</td>
</tr>
<tr>
<td>TRANSLATE TWO TO ONE</td>
<td>TRTO</td>
<td>RRE C E2 A SP</td>
<td>GM ST R1 R2 B991</td>
</tr>
<tr>
<td>TRANSLATE TWO TO TWO</td>
<td>TRTT</td>
<td>RRE C E2 A SP</td>
<td>GM ST R1 R2 B990</td>
</tr>
<tr>
<td>UNPACK</td>
<td>UNPK</td>
<td>SS A</td>
<td>ST B1 B2 F3</td>
</tr>
<tr>
<td>UNPACK ASCII</td>
<td>UNPKA</td>
<td>SS C E2 A SP</td>
<td>ST B1 B2 EA</td>
</tr>
<tr>
<td>UNPACK UNICODE</td>
<td>UNPKU</td>
<td>SS C E2 A SP</td>
<td>ST B1 B2 E2</td>
</tr>
<tr>
<td>UPDATE TREE</td>
<td>UPT</td>
<td>E C A SP II GM</td>
<td>ST I4 0102</td>
</tr>
</tbody>
</table>

**Explanation:**

¢ Causes serialization and checkpoint synchronization.
¢ Causes serialization and checkpoint synchronization when the M1 and R2 fields contain all ones and all zeros, respectively.
$ Causes serialization.
A Access exceptions for logical addresses.
A¹ Access exceptions; not all access exceptions may occur; see instruction description for details.
AI Access exceptions for instruction address.
B PER branch event.
B¹ B¹ field designates an access register in the access-register mode.
B² B² field designates an access register in the access-register mode.
BP B² field designates an access register when PSW bits 16 and 17 have the value 01 binary.
C Condition code is set.
Dd Decimal-operand data exception.
E E instruction format.
E2 Extended-translation facility 2.
E3 Extended-translation facility 3.
EX Execute exception.
FC Designation of access registers depends on the function code of the instruction.
G0 Instruction execution includes the implied use of general register 0.

*Figure 7-1 (Part 8 of 9). Summary of General Instructions*
**Explanation (Continued):**

GM Instruction execution includes the implied use of multiple general registers:
- General registers 1, 2, and 3 for COMPARE AND FORM CODEWORD.
- General registers 0 and 1 for COMPARE UNTIL SUBSTRING EQUAL and PERFORM LOCKED OPERATION.
- General registers 0 and 1 for COMPRESSION CALL, TRANSLATE ONE TO ONE, TRANSLATE TWO TO ONE, TRANSLATE TWO TO TWO
- General registers 1 and 2 for TRANSLATE AND TEST and TRANSLATE AND TEST REVERSED.
- General registers 0-5 for UPDATE TREE.

I1 Access register 1 is implicitly designated in the access-register mode.
I2 Access register 2 is implicitly designated in the access-register mode.
I4 Access register 4 is implicitly designated in the access-register mode.
IF Fixed-point-overflow exception.
II Interruptible instruction.
IK Fixed-point-divide exception.
L New condition code is loaded.
LD Long-displacement facility.
MO Monitor event.
MS Message-security assist.
N Instruction is new in z/Architecture as compared to ESA/390.
N3 Instruction is new in z/Architecture and has been added to ESA/390. Any RSY or RXY instructions still use the RSE or RXE format and 12-bit displacements in ESA/390.
R1 R1 field designates an access register in the access-register mode.
R2 R2 field designates an access register in the access-register mode.
R3 R3 field designates an access register in the access-register mode.
RI RI instruction format.
RIE RIE instruction format.
RIL RIL instruction format.
RM R1 field designates an access register in the access-register mode, and access-register 1 also is used in the access-register mode.
RR RR instruction format.
RRE RRE instruction format.
RS RS instruction format.
RSI RSI instruction format.
RXY RXY instruction format.
RSY RSY instruction format.
RX RX instruction format.
S S instruction format.
SI SI instruction format.
SIY SIY instruction format.
SP Specification exception.
SS SS instruction format.
ST PER storage-alteration event.
T Trace exceptions (includes trace table, addressing, and low-address protection).
U1 R1 field designates an access register unconditionally.
U2 R2 field designates an access register unconditionally.
UB R1 and R3 fields designate access registers unconditionally, and B2 field designates an access register in the access-register mode.

**Figure 7-1 (Part 9 of 9). Summary of General Instructions**
ADD

AR R₁, R₂ [RR]

┌────────┬────┬────┐
│ '1A'   │ R₁  │ R₂  │
└────────┴────┴────┘

0   8   12  15

AGR R₁, R₂ [RRE]

┌────────────────┬────────┬────┬────┐
│ 'B908' ////////// │ R₁    │ R₂  │
└────────────────┴────────┴────┴────┘

0   16   24   28  31

AGFR R₁, R₂ [RRE]

┌─────────────┬────┬────┬────┬────────┐
│ 'B918'     │ R₁  │ R₂  │ B₈  │ D₈    │
└─────────────┴────┴────┴────┴────────┘

0   16   24   28  31

The second operand is added to the first operand, and the sum is placed at the first-operand location. For ADD (AR, A, and AY), the operands and the sum are treated as 32-bit signed binary integers. For ADD (AGR, AG), they are treated as 64-bit signed binary integers. For ADD (AGFR, AGF), the second operand is treated as a 32-bit signed binary integer, and the first operand and the sum are treated as 64-bit signed binary integers.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

The displacement for A is treated as a 12-bit unsigned binary integer. The displacement for AY, AG, and AGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0  Result zero; no overflow
1  Result less than zero; no overflow
2  Result greater than zero; no overflow
3  Overflow

Program Exceptions:

• Access (fetch, operand 2 of A, AY, AG, and AGF only)
• Fixed-point overflow
• Operation (AY, if the long-displacement facility is not installed)

ADD HALFWORD

AH R₁, D₂(X₂, B₂) [RX]

┌────────┬────┬────┬────┬────────────┐
│ '4A'   │ R₁  │ X₂  │ B₂  │ D₂    │
└────────┴────┴────┴────┴────────────┘

0   8   12  16  20  31

AHY R₁, D₂(X₂, B₂) [RXY]

┌────────┬────┬────┬────┬──/─┬────────┬────────┐
│ 'E3'   │ R₁  │ X₂  │ B₂  │DL₂  │ DH₂  │ '5A'   │
└────────┴────┴────┴────┴──/─┴────────┴────────┘

0   8   12  16  20  32  40  47

AG R₁, D₂(X₂, B₂) [RXY]

┌────────┬────┬────┬────┬──/─┬────────┬────────┐
│ 'E3'   │ R₁  │ X₂  │ B₂  │DL₂  │ DH₂  │ '08'   │
└────────┴────┴────┴────┴──/─┴────────┴────────┘

0   8   12  16  20  32  40  47

AGF R₁, D₂(X₂, B₂) [RXY]

┌────────┬────┬────┬────┬──/─┬────────┬────────┐
│ 'E3'   │ R₁  │ X₂  │ B₂  │DL₂  │ DH₂  │ '18'   │
└────────┴────┴────┴────┴──/─┴────────┴────────┘

0   8   12  16  20  32  40  47

ADD HALFWORD IMMEDIATE

AHI R₁, I₂ [RI]

┌────────┬────┬────┐
│ 'A7'   │ R₁  │ 'A' │
│ I₂     │     │     │
└────────┴────┴────┘

0   8   12  16  31

AGHI R₁, I₂ [RI]

┌─────────┬────┬────┐
│ 'A7'    │ R₁  │ 'B' │
│ I₂      │     │     │
└─────────┴────┴────┘

0   8   12  16  31
The second operand is added to the first operand, and the sum is placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For ADD HALFWORD (AH, AHY) and ADD HALFWORD IMMEDIATE (AHl), the first operand and the sum are treated as 32-bit signed binary integers. For ADD HALFWORD IMMEDIATE (AGHI), they are treated as 64-bit signed binary integers.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

The displacement for AH is treated as a 12-bit unsigned binary integer. The displacement for AHY is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

- 0 Result zero; no overflow
- 1 Result less than zero; no overflow
- 2 Result greater than zero; no overflow
- 3 Overflow

**Program Exceptions:**

- Access (fetch, operand 2 of AH, AHY)
- Fixed-point overflow
- Operation (AHY, if the long-displacement facility is not installed)

**Programming Note:** An example of the use of the ADD HALFWORD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

---

**ADD LOGICAL**

<table>
<thead>
<tr>
<th>ALR</th>
<th>R1, R2</th>
<th>[RR]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'1E'</td>
<td>R1, R2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALGR</th>
<th>R1, R2</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B90A'</td>
<td>/---------/</td>
<td>R1, R2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALGFR</th>
<th>R1, R2</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B91A'</td>
<td>/---------/</td>
<td>R1, R2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AL</th>
<th>R1, D2(X2, B2)</th>
<th>[RX]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'5E'</td>
<td>R1, X2, B2, D2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALY</th>
<th>R1, D2(X2, B2)</th>
<th>[RXY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E3'</td>
<td>R1, X2, B2, DL2, DH2</td>
<td>'5E'</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALG</th>
<th>R1, D2(X2, B2)</th>
<th>[RXY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E3'</td>
<td>R1, X2, B2, DL2, DH2</td>
<td>'0A'</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALGF</th>
<th>R1, D2(X2, B2)</th>
<th>[RXY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E3'</td>
<td>R1, X2, B2, DL2, DH2</td>
<td>'1A'</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

The second operand is added to the first operand, and the sum is placed at the first-operand location. For ADD LOGICAL (ALR, AL, ALY), the operands and the sum are treated as 32-bit unsigned binary integers. For ADD LOGICAL (ALGR, ALG), they are treated as 64-bit unsigned binary integers. For ADD LOGICAL (ALGFR, ALGF) the second operand is treated as a 32-bit unsigned binary integer, and the first operand and the sum are treated as 64-bit unsigned binary integers.

The displacement for AL is treated as a 12-bit unsigned binary integer. The displacement for ALY, ALG, and ALGF is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

- 0 Result zero; no carry
- 1 Result not zero; no carry
- 2 Result zero; carry
- 3 Result not zero; carry
**Program Exceptions:**

- Access (fetch, operand 2 of AL, ALY, ALG, and ALGF only)
- Operation (ALY, if the long-displacement facility is not installed)

### ADD LOGICAL WITH CARRY

**ALCR**  
\( R_1, R_2 \)  
\[ [RRE] \]

\[
\begin{array}{c|ccccc|c}
'B998' & / / / / / / / & R_1 & R_2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]

**ALCG R**  
\( R_1, R_2 \)  
\[ [RRE] \]

\[
\begin{array}{c|ccccc|c}
'B988' & / / / / / / / & R_1 & R_2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]

**ALC**  
\( R_1, D_2(X_2, B_2) \)  
\[ [RXY] \]

\[
\begin{array}{c|ccccc|c}
'E3' & R_1 & X_2 & B_2 & D L_2 & D H_2 & '98' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47 \\
\end{array}
\]

**ALCG**  
\( R_1, D_2(X_2, B_2) \)  
\[ [RXY] \]

\[
\begin{array}{c|ccccc|c}
'E3' & R_1 & X_2 & B_2 & D L_2 & D H_2 & '88' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47 \\
\end{array}
\]

The second operand and the carry are added to the first operand, and the sum is placed at the first-operand location. For ADD LOGICAL WITH CARRY (ALCR, ALC), the operands, the carry, and the sum are treated as 32-bit unsigned binary integers. For ADD LOGICAL WITH CARRY (ALCGR, ALCG), they are treated as 64-bit unsigned binary integers.

**Resulting Condition Code:**

0  Result zero; no carry
1  Result not zero; no carry
2  Result zero; carry
3  Result not zero; carry

**Program Exceptions:**

- Access (fetch, operand 2 of ALC and ALCG only)

### Programming Notes:

1. A carry is represented by a one value of bit 18 of the current PSW. Bit 18 is the leftmost bit of the two-bit condition code in the PSW. Bit 18 is set to one by an execution of an ADD LOGICAL or ADD LOGICAL WITH CARRY instruction that produces a carry out of bit position 0 of the result.

2. ADD and ADD LOGICAL may provide better performance than ADD LOGICAL WITH CARRY, depending on the model.

### AND

**NR**  
\( R_1, R_2 \)  
\[ [RR] \]

\[
\begin{array}{c|cc|c}
'14' & R_1 & R_2 \\
0 & 8 & 12 & 15 \\
\end{array}
\]

**NGR**  
\( R_1, R_2 \)  
\[ [RRE] \]

\[
\begin{array}{c|ccccc|c}
'B980' & / / / / / / / & R_1 & R_2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]

**N**  
\( R_1, D_2(X_2, B_2) \)  
\[ [RX] \]

\[
\begin{array}{c|ccccc|c}
'54' & R_1 & X_2 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47 \\
\end{array}
\]

**NY**  
\( R_1, D_2(X_2, B_2) \)  
\[ [RXY] \]

\[
\begin{array}{c|ccccc|c}
'E3' & R_1 & X_2 & B_2 & D L_2 & D H_2 & '54' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47 \\
\end{array}
\]

**NG**  
\( R_1, D_2(X_2, B_2) \)  
\[ [RXY] \]

\[
\begin{array}{c|ccccc|c}
'E3' & R_1 & X_2 & B_2 & D L_2 & D H_2 & '80' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47 \\
\end{array}
\]

**NI**  
\( D_1(B_1), I_2 \)  
\[ [SI] \]

\[
\begin{array}{c|ccccc|c}
'94' & I_2 & B_1 & D_1 \\
0 & 8 & 16 & 20 & 31 \\
\end{array}
\]
The AND of the first and second operands is placed at the first-operand location.

The connective AND is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit positions in both operands contain ones; otherwise, the result bit is set to zero.

For AND (NC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

For AND (NI, NIY), the first operand is one byte in length, and only one byte is stored.

For AND (NR, N, NY), the operands are 32 bits, and for AND (NGR, NG), they are 64 bits.

The displacements for N, NI, and both operands of NC are treated as 12-bit unsigned binary integers. The displacement for NY, NIY, and NG is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0  Result zero
1  Result not zero
2  --
3  --

Program Exceptions:

- Access (fetch, operand 2, N, NY, NG, and NC; fetch and store, operand 1, NI, NIY, and NC)
- Operation (NY and NIY, if the long-displacement facility is not installed)

Chapter 7. General Instructions 7-21

Programming Notes:

1. An example of the use of the AND instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The AND instruction may be used to set a bit to zero.

3. Accesses to the first operand of AND (NI) and AND (NC) consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, the instruction AND cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in “Multiprogramming and Multiproducting Examples” on page A-43.

AND IMMEDIATE

<table>
<thead>
<tr>
<th>Code</th>
<th>R1, I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIHH</td>
<td>'A5' R1 '4' I2</td>
</tr>
<tr>
<td>0 08 12 16 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>R1, I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIHL</td>
<td>'A5' R1 '5' I2</td>
</tr>
<tr>
<td>0 08 12 16 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>R1, I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NILH</td>
<td>'A5' R1 '6' I2</td>
</tr>
<tr>
<td>0 08 12 16 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>R1, I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>NILL</td>
<td>'A5' R1 '7' I2</td>
</tr>
<tr>
<td>0 08 12 16 31</td>
<td></td>
</tr>
</tbody>
</table>

The second operand is ANDed with bits of the first operand, and the result replaces those bits of the first operand. The remainder of the first operand remains unchanged.
For each instruction, the bits of the first operand that are ANDed with the second operand and then replaced are as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ANDed and Replaced</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIHH</td>
<td>0-15</td>
</tr>
<tr>
<td>NIHL</td>
<td>16-31</td>
</tr>
<tr>
<td>NILH</td>
<td>32-47</td>
</tr>
<tr>
<td>NILL</td>
<td>48-63</td>
</tr>
</tbody>
</table>

The connective AND is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit positions in both operands contain ones; otherwise, the result bit is set to zero.

**Resulting Condition Code:**

0  Sixteen-bit result zero
1  Sixteen-bit result not zero
2  --
3  --

**Program Exceptions:** None.

**BRANCH AND LINK**

**BALR**  
R1,R2  [RR]

<table>
<thead>
<tr>
<th>'05'</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

**BAL**  
R1,D2(X2,B2)  [RX]

<table>
<thead>
<tr>
<th>'45'</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

The link information in the 24-bit addressing mode consists of the instruction-length code (ILC), the condition code (CC), the program-mask bits, and the rightmost 24 bits of the updated instruction address, arranged in bit positions 32-63 of the first-operand location in the following format:

<table>
<thead>
<tr>
<th>IL</th>
<th>C</th>
<th>Prog</th>
<th>Mask</th>
<th>Instruction Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>34</td>
<td>36</td>
<td>40</td>
<td>63</td>
</tr>
</tbody>
</table>

The instruction-length code is 1 or 2.

The link information in the 31-bit addressing mode consists of bit 32 of the PSW, the basic-addressing-mode bit (always a one) and the rightmost 31 bits of the updated instruction address, arranged in bit positions 32-63 of the first-operand location in the following format:

<table>
<thead>
<tr>
<th>1</th>
<th>Instruction Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>63</td>
</tr>
</tbody>
</table>

In the 32-bit addressing mode, bits 0-31 of the first-operand location remain unchanged.

The link information in the 64-bit addressing mode consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R2 are used to generate the branch address; however, when the R2 field is zero, the operation is performed without branching. The branch address is computed before general register R1 is changed.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Trace (R2 field nonzero, BALR only)

**Programming Notes:**

1. An example of the use of the BRANCH AND LINK instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. When the R2 field in the RR format is zero, the link information is loaded without branching.

3. The BRANCH AND LINK instruction (BAL and BALR) is provided for compatibility purposes. It is recommended that, where possible, the BRANCH AND SAVE instruction (BAS and
BASR), BRANCH RELATIVE AND SAVE, or BRANCH RELATIVE AND SAVE LONG be used and BRANCH AND LINK avoided, since the latter places nonzero information in bit positions 32-39 of the link register in the 24-bit addressing mode, which may lead to problems. Additionally, in the 24-bit addressing mode, BRANCH AND LINK may be slower than the other instructions because BRANCH AND LINK must construct the ILC, condition code, and program mask to be placed in bit positions 32-39 of the link register.

4. The condition-code and program-mask information, which is provided in the leftmost byte of the link information only in the 24-bit addressing mode, can be obtained in any addressing mode by means of the INSERT PROGRAM MASK instruction.

### BRANCH AND SAVE

\[
\text{BASR } R_1, R_2 \quad [\text{RR}]
\]

\[
\begin{array}{cccc}
\text{0} & \text{8} & \text{12} & \text{15}
\end{array}
\]

\[
\text{BAS } R_1, D_2(X_2, B_2) \quad [\text{RX}]
\]

\[
\begin{array}{cccc}
\text{0} & \text{8} & \text{12} & \text{16} & \text{20} & \text{31}
\end{array}
\]

Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged.

In the 64-bit addressing mode, the link information consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register R2 are used to generate the branch address; however, when the R2 field is zero, the operation is performed without branching. The branch address is computed before general register R1 is changed.

### Condition Code:
The code remains unchanged.

### Program Exceptions:
- Trace (R2 field nonzero, BASR only)

### Programming Notes:
1. An example of the use of the BRANCH AND SAVE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. The BRANCH AND SAVE instruction (BAS and BASR) is intended to be used for linkage to programs known to be in the same addressing mode as the caller. This instruction should be used in place of the BRANCH AND LINK instruction (BAL and BALR). See the programming notes on pages 5-12 and 5-18 in the section "Subroutine Linkage without the Linkage Stack" for a detailed discussion of this and other linkage instructions. See also the programming note under BRANCH AND LINK for a discussion of the advantages of the BRANCH AND SAVE instruction.

### BRANCH AND SAVE AND SET MODE

\[
\text{BASSM } R_1, R_2 \quad [\text{RR}]
\]

\[
\begin{array}{cccc}
\text{0} & \text{8} & \text{12} & \text{15}
\end{array}
\]

Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, if the R2 field is nonzero, the addressing-mode bits and instruction address in the PSW are replaced as specified by the second operand.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and
the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged. In the 64-bit addressing mode, the link information is bits 64-126 of the PSW with a one appended on the right, placed in bit positions 0-63 of the first-operand location.

The contents of general register \( R_z \) specify the new addressing mode and designate the branch address; however, when the \( R_z \) field is zero, the operation is performed without branching and without setting either addressing-mode bit.

When the contents of general register \( R_z \) are used and bit 63 of the register is zero, bit 31 of the current PSW, the extended-addressing-mode bit, is set to zero, bit 32 of the register specifies the new basic addressing mode and replaces bit 32 of the PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The branch address replaces the instruction address in the PSW.

When the contents of general register \( R_z \) are used and bit 63 of the register is one, the following occurs. Bits 31 and 32 of the current PSW are set to one, the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new extended addressing mode, and the branch address replaces the instruction address in the PSW. Bit 63 of the register remains one. However, if \( R_z \) is the same as \( R_1 \), the results in the designated general register are as specified for the \( R_1 \) register.

The new value for the PSW is computed before general register \( R_1 \) is changed.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Trace (\( R_z \) field nonzero)

**Programming Notes:**
1. An example of the use of the BRANCH AND SAVE AND SET MODE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. BRANCH AND SAVE AND SET MODE is intended to be the principal calling instruction to subroutines which may operate in a different addressing mode from that of the caller. See the programming notes on pages 5-12 and 5-18 in the section “Subroutine Linkage without the Linkage Stack” for a detailed discussion of this and other linkage instructions.

3. An old 24-bit or 31-bit program can use BRANCH AND SAVE AND SET MODE to call a new 64-bit program without any change, provided that bits 0-31 of general register \( R_z \) are all zeros. The old program can load into bit positions 32-63 of general register \( R_z \) a four-byte address constant, which is provided from outside the program, in which bit 63 in the register (bit 31 of the constant in storage) either is or is not one. If the addressing mode is not changed to the 64-bit mode by the execution of the BRANCH AND SAVE AND SET MODE instruction, or even if it is, the called program can set the 64-bit mode by issuing a SET ADDRESSING MODE (SAM64) instruction.

4. See the programming notes on page 5-12 (under “Simple Branch Instructions”).

**BRANCH AND SET MODE**

\[
\text{BSM } R_1, R_z \quad [\text{RR}]
\]

\[
\begin{array}{ccc}
0 & 8 & 12 & 15 \\
\end{array}
\]

In the 24-bit or 31-bit addressing mode, bit 32 of the current PSW, the basic-addressing-mode bit, is inserted into bit position 32 of the first operand, and bits 0-31 and 33-63 of the operand remain unchanged. In the 64-bit addressing mode, a one is inserted into bit position 63 of the first operand, and bits 0-62 of the operand remain unchanged. Subsequently, the addressing-mode bits and instruction address in the PSW are replaced as specified by the second operand. The action associated with an operand is not performed if the associated \( R \) field is zero.

The contents of general register \( R_z \) specify the new addressing mode and designate the branch address; however, when the \( R_z \) field is zero, the operation is performed without branching and without setting either addressing-mode bit.
When the contents of general register \( R_2 \) are used and bit 63 of the register is zero, bit 31 of the current PSW, the extended-addressing-mode bit, is set to zero, bit 32 of the register specifies the new basic addressing mode and replaces bit 32 of the PSW, and the branch address is generated from the contents of the register under the control of the new addressing mode. The branch address replaces the instruction address in the PSW.

When the contents of general register \( R_2 \) are used and bit 63 of the register is one, the following occurs. Bits 31 and 32 of the current PSW are set to one, the branch address is generated from the contents of the register, except with bit 63 of the register treated as a zero, under the control of the new extended addressing mode, and the branch address replaces the instruction address in the PSW. Bit 63 of the register remains one. However, if \( R_2 \) is the same as \( R_1 \), the results in the designated general register are as specified for the \( R_1 \) register.

The new value for the PSW is computed before general register \( R_1 \) is changed.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Trace

**Programming Notes:**
1. An example of the use of the BRANCH AND SET MODE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. BRANCH AND SET MODE with an \( R_1 \) field of zero is intended to be the standard return instruction in a program entered by means of BRANCH AND SAVE AND SET MODE. It can also be the return instruction in a program entered in the 24-bit or 31-bit addressing mode by means of BRANCH AND SAVE, BRANCH RELATIVE AND SAVE, or BRANCH RELATIVE AND SAVE LONG. BRANCH AND SET MODE with a nonzero \( R_1 \) field is intended to be used in a "glue module" to connect either old 24-bit programs and newer programs that are executed in the 31-bit addressing mode or old 24-bit or 31-bit programs and new programs that are executed in the 64-bit addressing mode. See the programming notes on pages 5-12 and 5-18 in the section "Subroutine Linkage without the Linkage Stack" for a detailed discussion of this and other linkage instructions.

**BRANCH ON CONDITION**

\[
\text{BCR } M_1, R_2 \quad [\text{RR}]
\]
\[
\begin{array}{cccc}
0 & 8 & 12 & 15 \\
\end{array}
\]

\[
\text{BC } M_1, D_2(X_2, B_2) \quad [\text{RX}]
\]
\[
\begin{array}{cccccc}
0 & 8 & 12 & 16 & 20 & 31 \\
\end{array}
\]

The instruction address in the current PSW is replaced by the branch address if the condition code has one of the values specified by \( M_1 \); otherwise, normal instruction sequencing proceeds with the updated instruction address.

In the RX format, the second-operand address is used as the branch address. In the RR format, the contents of general register \( R_2 \) are used to generate the branch address; however, when the \( R_2 \) field is zero, the operation is performed without branching.

The \( M_1 \) field is used as a four-bit mask. The four condition codes (0, 1, 2, and 3) correspond, left to right, with the four bits of the mask, as follows:

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Instruction Bit No. of Mask</th>
<th>Mask Position Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

The current condition code is used to select the corresponding mask bit. If the mask bit selected by the condition code is one, the branch is successful. If the mask bit selected is zero, normal instruction sequencing proceeds with the next sequential instruction.

When the \( M_1 \) and \( R_2 \) fields of BRANCH ON CONDITION (BCR) are all ones and all zeros, respec-
tively, a serialization and checkpoint-synchronization function is performed.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**Programming Notes:**

1. An example of the use of the BRANCH ON CONDITION instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. When a branch is to depend on more than one condition, the pertinent condition codes are specified in the mask as the sum of their mask position values. A mask of 12, for example, specifies that a branch is to be made when the condition code is 0 or 1.

3. When all four mask bits are zeros or when the R₂ field in the RR format contains zero, the branch instruction is equivalent to a no-operation. When all four mask bits are ones, that is, the mask value is 15, the branch is unconditional unless the R₂ field in the RR format is zero.

4. Execution of BCR 15,0 (that is, an instruction with a value of 07F0 hex) may result in significant performance degradation. To ensure optimum performance, the program should avoid use of BCR 15,0 except in cases when the serialization or checkpoint-synchronization function is actually required.

5. Note that the relation between the RR and RX formats in branch-address specification is not the same as in operand-address specification. For branch instructions in the RX format, the branch address is the address specified by X₂, B₂, and D₂; in the RR format, the branch address is contained in the register designated by R₂. For operands, the address specified by X₂, B₂, and D₂ is the operand address, but the register designated by R₂ contains the operand, not the operand address.

---

**BRANCH ON COUNT**

### BCTR R₁,R₂ [RR]

```
'06'  R₁  R₂
0  8  12  15
```

### BCTGR R₁,R₂ [RRE]

```
'B946'  ///////  R₁  R₂
0  16  24  28  31
```

### BCT R₁,D₂(X₂,B₂) [RX]

```
'46'  R₁  X₂  B₂  D₂
0  8  12  16  20  31
```

### BCTG R₁,D₂(X₂,B₂) [RXY]

```
'E3'  R₁  X₂  B₂  DL₂  DH₂  '46'
0  8  12  16  20  40  47
```

A one is subtracted from the first operand, and the result is placed at the first-operand location. For BRANCH ON COUNT (BCT, BCTR), the first operand and result are treated as 32-bit binary integers, with overflow ignored. For BRANCH ON COUNT (BCTG, BCTGR), the first operand and result are treated as 64-bit binary integers, with overflow ignored. When the result is zero, normal instruction sequencing proceeds with the updated instruction address. When the result is not zero, the instruction address in the current PSW is replaced by the branch address.

In the RX or RXY format, the second-operand address is used as the branch address. In the RR or RRE format, the contents of general register R₂ are used to generate the branch address; however, when the R₂ field is zero, the operation is performed without branching. The branch address is generated before general register R₁ is changed.

**Condition Code:** The code remains unchanged.
Program Exceptions: None.

Programming Notes:

1. An example of the use of the BRANCH ON COUNT instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The first operand and result can be considered as either signed or unsigned binary integers since the result of a binary subtraction is the same in both cases.

3. An initial count of one results in zero, and no branching takes place; an initial count of zero results in -1 and causes branching to be performed; an initial count of -1 results in -2 and causes branching to be performed; and so on. In a loop, branching takes place each time the instruction is executed until the result is again zero. Note that for BCT or BCTR, because of the number range, an initial count of \(-2^{31}\) results in a positive value of \(2^{31} - 1\), or, for BCTG or BCTGR, an initial count of \(-2^{63}\) results in a positive value of \(2^{63} - 1\).

4. Counting is performed without branching when the \(R_2\) field in the RR or RRE format contains zero.

### BRANCH ON INDEX HIGH

**BXH**  \(R_1, R_3, D_2(B_2)\)  [RS]

<table>
<thead>
<tr>
<th>()</th>
<th>(R_1)</th>
<th>(R_3)</th>
<th>(B_2)</th>
<th>(D_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>'86'</td>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

**BXHG**  \(R_1, R_3, D_2(B_2)\)  [RSY]

<table>
<thead>
<tr>
<th>()</th>
<th>()</th>
<th>()</th>
<th>()</th>
<th>()</th>
<th>()</th>
<th>()</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EB'</td>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

### BRANCH ON INDEX LOW OR EQUAL

**BXLE**  \(R_1, R_3, D_2(B_2)\)  [RS]

<table>
<thead>
<tr>
<th>()</th>
<th>(R_1)</th>
<th>(R_3)</th>
<th>(B_2)</th>
<th>(D_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>'87'</td>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

An increment is added to the first operand, and the sum is compared with a compare value. The result of the comparison determines whether branching occurs. Subsequently, the sum is placed at the first-operand location. The second-operand address is used as a branch address. The \(R_3\) field designates registers containing the increment and the compare value.

For **BRANCH ON INDEX HIGH**, when the sum is high, the instruction address in the current PSW is replaced by the branch address. When the sum is low or equal, normal instruction sequencing proceeds with the updated instruction address.

For **BRANCH ON INDEX LOW OR EQUAL**, when the sum is low or equal, the instruction address in the current PSW is replaced by the branch address. When the sum is high, normal instruction sequencing proceeds with the updated instruction address.

When the \(R_3\) field is even, it designates a pair of registers; the contents of the even and odd registers of the pair are used as the increment and the compare value, respectively. When the \(R_3\) field is odd, it designates a single register, the contents of which are used as both the increment and the compare value.

For purposes of the addition and comparison, all operands and results are treated as 32-bit signed binary integers for BXH and BXLE or as 64-bit signed binary integers for BXHG and BXLEG. Overflow caused by the addition is ignored.

The original contents of the compare-value register are used as the compare value even when that register is also specified to be the first-operand location. The branch address is generated before general register \(R_1\) is changed.

The sum is placed at the first-operand location, regardless of whether the branch is taken.

**Condition Code:** The code remains unchanged.
**Program Exceptions:** None.

**Programming Notes:**

1. Several examples of the use of the BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL instructions are given in Appendix A, "Number Representation and Instruction-Use Examples."

2. The word “index” in the names of these instructions indicates that one of the major purposes is the incrementing and testing of an index value. The increment, being a signed binary integer, may be used to increase or decrease the value in general register R1 by an arbitrary amount, subject to the limit of the integer size.

3. Care must be taken in the 31-bit addressing mode when a data area in storage is at the rightmost end of a 31-bit address space and a BRANCH ON INDEX HIGH (BXH) or BRANCH ON INDEX LOW OR EQUAL (BXLE) instruction is used to step upward through the data. Since the addition and comparison operations performed during the execution of these instructions treat the operands as 32-bit signed binary integers, the value following $2^{31} - 1$ is not $2^{31}$, which cannot be represented in that format, but $-2^{31}$. The instruction does not provide an indication of such overflow. Consequently, some common looping techniques based on the use of these instructions do not work when a data area ends at address $2^{31} - 1$. This problem is illustrated in a BRANCH ON INDEX LOW OR EQUAL example in Appendix A, "Number Representation and Instruction-Use Examples." A similar caution applies in the 64-bit addressing mode when data is at the end of a 64-bit address space and BRANCH ON INDEX HIGH (BXHG) or BRANCH ON INDEX LOW OR EQUAL (BXLEG) is used.

**BRANCH RELATIVE AND SAVE LONG**

```
BRASL  R1, I2  [RIL]
```

Information from the current PSW, including the updated instruction address, is saved as link information at the first-operand location. Subsequently, the instruction address in the PSW is replaced by the branch address.

In the 24-bit or 31-bit addressing mode, the link information is bits 32 and 97-127 of the PSW, consisting of the basic-addressing-mode bit and the rightmost 31 bits of the updated instruction address. The link information is placed in bit positions 32 and 33-63, respectively, of the first-operand location, and bits 0-31 of the location remain unchanged.

In the 64-bit addressing mode, the link information consists of the updated instruction address, placed in bit positions 0-63 of the first-operand location.

The contents of the I2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**Programming Notes:**

1. The operation is the same as that of the BRANCH AND SAVE (BAS) instruction except for the means of specifying the branch address. An example of the use of BRANCH AND SAVE is given in Appendix A.

2. The BRANCH RELATIVE AND SAVE and BRANCH RELATIVE AND SAVE LONG instructions, like the BRANCH AND SAVE instruction, are intended to be used for linkage to programs known to be in the same addressing mode as the caller. These instructions should be used in place of the BRANCH AND LINK instruction (BAL and BALR). See the programming notes on pages 5-12 and 5-18 in the section "Subroutine"
Linkage without the Linkage Stack” for a
detailed discussion of these and other linkage
instructions. See also the programming note
under BRANCH AND LINK for a discussion of
the advantages of the BRANCH RELATIVE
AND SAVE, BRANCH RELATIVE AND SAVE
LONG, and BRANCH AND SAVE instructions.

3. When the instruction is the target of
EXECUTE, the branch is relative to the target
address; see “Branch-Address Generation” on
page 5-10.

BRANCH RELATIVE ON
CONDITION

BRC  M₁,₁₂  [RI]

<table>
<thead>
<tr>
<th>'A7'</th>
<th>M₁</th>
<th>'4'</th>
<th>₁₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

BRANCH RELATIVE ON
CONDITION LONG

BRCL  M₁,₁₂  [RIL]

<table>
<thead>
<tr>
<th>'C0'</th>
<th>M₁</th>
<th>'4'</th>
<th>₁₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

The instruction address in the current PSW is
replaced by the branch address if the condition
code has one of the values specified by M₁; other-
wise, normal instruction sequencing proceeds with
the updated instruction address.

The contents of the ₁₂ field are a signed binary
integer specifying the number of halfwords that is
added to the address of the instruction to generate
the branch address.

The M₁ field is used as a four-bit mask. The four
condition codes (0, 1, 2, and 3) correspond, left to
right, with the four bits of the mask, as follows:

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Instruction Bit No. of Mask</th>
<th>Mask Position Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

The current condition code is used to select the
corresponding mask bit. If the mask bit selected
by the condition code is one, the branch is suc-
cessful. If the mask bit selected is zero, normal
instruction sequencing proceeds with the next
sequential instruction.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**Programming Notes:**

1. The operation is the same as that of the
BRANCH ON CONDITION instruction except
for the means of specifying the branch
address. An example of the use of BRANCH
ON CONDITION is given in Appendix A.

2. When a branch is to depend on more than
one condition, the pertinent condition codes
are specified in the mask as the sum of their
mask position values. A mask of 12, for
example, specifies that a branch is to be
made when the condition code is 0 or 1.

3. When all four mask bits are zeros, the branch
instruction is equivalent to a no-operation.
When all four mask bits are ones, that is, the
mask value is 15, the branch is unconditional.

4. When the instruction is the target of
EXECUTE, the branch is relative to the target
address; see “Branch-Address Generation” on
page 5-10.

BRANCH RELATIVE ON COUNT

BRCT  R₁,₁₂  [RI]

<table>
<thead>
<tr>
<th>'A7'</th>
<th>R₁</th>
<th>'6'</th>
<th>₁₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

Chapter 7. General Instructions 7-29
A one is subtracted from the first operand, and the result is placed at the first-operand location. For BRANCH RELATIVE ON COUNT (BRCT), the first operand and result are treated as 32-bit binary integers, with overflow ignored. For BRANCH RELATIVE ON COUNT (BRCTG), the first operand and result are treated as 64-bit binary integers, with overflow ignored. When the result is zero, normal instruction sequencing proceeds with the updated instruction address. When the result is not zero, the instruction address in the current PSW is replaced by the branch address.

The contents of the I2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**Programming Notes:**

1. The operation is the same as that of the BRANCH ON COUNT instruction except for the means of specifying the branch address. An example of the use of BRANCH ON COUNT is given in Appendix A.

2. The first operand and result can be considered as either signed or unsigned binary integers since the result of a binary subtraction is the same in both cases.

3. An initial count of one results in zero, and no branching takes place; an initial count of zero results in -1 and causes branching to be executed; an initial count of -1 results in -2 and causes branching to be executed; and so on. In a loop, branching takes place each time the instruction is executed until the result is again zero. Note that for BRCT, because of the number range, an initial count of \(-2^{31}\) results in a positive value of \(2^{31} - 1\), or, for BRCTG, an initial count of \(-2^{63}\) results in a positive value of \(2^{63} - 1\).

4. When the instruction is the target of EXECUTE, the branch is relative to the target address; see “Branch-Address Generation” on page 5-10.

**BRANCH RELATIVE ON INDEX HIGH**

BRXH R1,R3,I2 [RSI]

```
+-------------+---+---+---------------+
| '84' | R1 | R3 | I2            |
+-------------+---+---+---------------+
```

```
0  8 12 16 31
```

BRXHG R1,R3,I2 [RIE]

```
+-------------+---+---+---------------+
| 'EC' | R1 | R3 | I2 ////////////// |
+-------------+---+---+---------------+
```

```
0  8 12 16 32 40 47
```

**BRANCH RELATIVE ON INDEX LOW OR EQUAL**

BRXLE R1,R3,I2 [RSI]

```
+-------------+---+---+---------------+
| '85' | R1 | R3 | I2            |
+-------------+---+---+---------------+
```

```
0  8 12 16 31
```

BRXLG R1,R3,I2 [RIE]

```
+-------------+---+---+---------------+
| 'EC' | R1 | R3 | I2 ////////////// |
+-------------+---+---+---------------+
```

```
0  8 12 16 32 40 47
```

An increment is added to the first operand, and the sum is compared with a compare value. The result of the comparison determines whether branching occurs. Subsequently, the sum is placed at the first-operand location. The R3 field designates registers containing the increment and the compare value.

The contents of the I2 field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the branch address.

For BRANCH RELATIVE ON INDEX HIGH, when the sum is high, the instruction address in the current PSW is replaced by the branch address. When the sum is low or equal, normal instruction sequencing proceeds with the updated instruction address.
For BRANCH RELATIVE ON INDEX LOW OR EQUAL, when the sum is low or equal, the instruction address in the current PSW is replaced by the branch address. When the sum is high, normal instruction sequencing proceeds with the updated instruction address.

When the R3 field is even, it designates a pair of registers; the contents of the even and odd registers of the pair are used as the increment and the compare value, respectively. When the R3 field is odd, it designates a single register, the contents of which are used as both the increment and the compare value.

For purposes of the addition and comparison, all operands and results are treated as 32-bit signed binary integers for BRXH and BRXLE or as 64-bit signed binary integers for BRXHG and BRXLG. Overflow caused by the addition is ignored.

The original contents of the compare-value register are used as the compare value even when that register is also specified to be the first-operand location.

The sum is placed at the first-operand location, regardless of whether the branch is taken.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**Programming Notes:**

1. The operations are the same as those of the BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL instructions except for the means of specifying the branch address. Several examples of the use of BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL are given in Appendix A.

2. The word “index” in the names of these instructions indicates that one of the major purposes is the incrementing and testing of an index value. The increment, being a signed binary integer, may be used to increase or decrease the value in general register R1 by an arbitrary amount.

3. Care must be taken in the 31-bit addressing mode when a data area in storage is at the rightmost end of an address space and a BRANCH RELATIVE ON INDEX HIGH (BRXH) or BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLE) instruction is used to step upward through the data. Since the addition and comparison operations performed during the execution of these instructions treat the operands as 32-bit signed binary integers, the value following \(2^{31} - 1\) is not \(2^{31}\), which cannot be represented in that format, but \(-2^{31}\). The instruction does not provide an indication of such overflow. Consequently, some common looping techniques based on the use of these instructions do not work when a data area ends at address \(2^{31} - 1\). This problem is illustrated in a BRANCH ON INDEX LOW OR EQUAL example in Appendix A. A similar caution applies in the 64-bit addressing mode when data is at the end of a 64-bit address space and BRANCH RELATIVE ON INDEX HIGH (BRXHG) or BRANCH RELATIVE ON INDEX LOW OR EQUAL (BRXLG) is used.

4. When the instruction is the target of EXECUTE, the branch is relative to the target address; see "Branch-Address Generation" on page 5-10.

**CHECKSUM**

\[
\begin{array}{|c|c|c|c|c|}
\hline
'8241' & // ////////// & R1 & R2 \\
\hline
0 & 16 & 24 & 28 & 31 \\
\hline
\end{array}
\]

Successive four-byte elements of the second operand are added to the first operand in bit positions 32-63 of general register R1 to form a 32-bit checksum in those bit positions. The first operand and the four-byte elements are treated as 32-bit unsigned binary integers. After each addition of an element, a carry out of bit position 32 of the first operand is added to bit position 63 of the first operand. Bits 0-31 of general register R1 always remain unchanged. If the second operand is not a multiple of four bytes, its last one, two, or three bytes are treated as appended on the right with the number of all-zeros bytes needed to form a four-byte element. The four-byte elements are added to the first operand until either the entire second operand or a CPU-determined amount of the second operand has been processed. The result is indicated in the condition code.
The $R_z$ field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the second operand is specified by the contents of the $R_z$ general register. The number of bytes in the second-operand location is specified by the 32-bit or 64-bit unsigned binary integer in the $R_z + 1$ general register.

The handling of the address in general register $R_z$ and the length in general register $R_z + 1$ is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register $R_z$ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the register constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of the register constitute the address. In the 24-bit or 31-bit addressing mode, the length is a 32-bit unsigned binary integer in bit positions 32-63 of general register $R_z + 1$, and the contents of bit positions 0-31 are ignored. In the 64-bit addressing mode, the length is a 64-bit unsigned binary integer in the register.

The addition of second-operand four-byte elements to the first operand proceeds left to right, four-byte element by four-byte element, and ends as soon as (1) the entire second operand has been processed or (2) a lesser CPU-determined amount of the second operand has been processed. In either case, the result in bit positions 32-63 of general register $R_z$ is a 32-bit checksum for the part of the second operand that has been processed. When the second operand is not a multiple of four bytes, the final second-operand bytes in excess of a multiple of four are conceptually appended on the right with an appropriate number of all-zeros bytes to form the final four-byte element.

If the operation ends because the entire second operand has been processed, the condition code is set to 0. If the operation ends because a lesser CPU-determined amount of the second operand has been processed, the condition code is set to 3. When the operation is to end with a setting of condition code 3, any carry out of bit position 32 of the first operand is added to bit position 63 of the first operand before the operation ends.

At the completion of the operation, the 32-bit or 64-bit operand-length field in the $R_z + 1$ register is decremented by the number of actual second-operand bytes added to the first operand (not including any conceptually appended all-zeros bytes), and the address in the $R_z$ register is incremented by the same number. Thus, the the 32-bit or 64-bit operand-length field contains a zero value if the condition code is set to 0, or it contains a nonzero value if the condition code is set to 3. In the 24-bit or 31-bit addressing mode, bits 0-31 of the $R_z + 1$ register always remain unchanged.

When condition code 3 is set, the general registers used by the instruction have been set so that the remainder of the second operand can be processed by simply branching back to reexecute the instruction.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed. The minimum amount is four bytes or the number of bytes specified in the $R_z + 1$ general register, whichever is smaller.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general register $R_z$ may be set to zeros or may remain unchanged, even when the initial length in register $R_z + 1$ is zero. Bits 0-31 of general register $R_z$ remain unchanged.

When the $R_z$ register is the same register as the $R_z$ or $R_z + 1$ register, the results are unpredictable.

Access exceptions for the portion of the second operand to the right of the last byte processed may or may not be recognized. For a second operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

Access exceptions are not recognized if the $R_z$ field is odd. When the length of the second operand is zero, no access exceptions are recognized.
**Resulting Condition Code:**

0  Entire second operand processed  
1  --  
2  --  
3  CPU-determined amount of second operand processed  

**Program Exceptions:**

- Access (fetch, operand 2)  
- Specification  

**Programming Notes:**

1. The initial contents of bit positions 32-63 of the R

   register contribute to the 32-bit checksum. The program normally should set those contents to all zeros before issuing the CHECKSUM instruction.  

2. A 16-bit checksum is used in, for example, the TCP/IP application. The following program can be executed after the CHECKSUM instruction to produce a 16-bit checksum from the 32-bit checksum in bit positions 32-63 of general register R. The program is annotated to show the contents of bit positions 32-63 of the R and R + 1 registers after the execution of each instruction. The contents of bit positions 32-63 of the R register are represented as A,B, meaning the value A in bit positions 32-47 and the value B in bit positions 48-63. The value C is a carry from A + B. Note that bit positions 32-63 of register R + 1 are known to contain all zeros when CHECKSUM has set condition code 0.  

<table>
<thead>
<tr>
<th>Program</th>
<th>R2 Bits</th>
<th>R2+1 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>R2,R1</td>
<td>A,B</td>
</tr>
<tr>
<td>SRDL</td>
<td>R2,16</td>
<td>0,A</td>
</tr>
<tr>
<td>ALR</td>
<td>R2,R2+1</td>
<td>B,A</td>
</tr>
<tr>
<td>ALR</td>
<td>R2,R1</td>
<td>A+B+C,A+B</td>
</tr>
<tr>
<td>SRL</td>
<td>R2,16</td>
<td>0,A+B+C</td>
</tr>
</tbody>
</table>

3. The CHECKSUM instruction may be used in computing hash values as illustrated in the following programming example. The variable KEY contains a string to be mapped into a slot in a hash table. The variable SIZE is a prime number designating the size of the hash table. The value of SIZE is determined by (a) the number of strings to be hashed into the table divided by the acceptable number of hash collisions, and (b) a value that is not too close to a power of two. Following the DIVIDE (D) instruction, the remainder in register 0 represents the resulting hash value.  

   SR 1,1 Zero accumulator  
   LA 2,KEY Point to string  
   LA 3,L'KEY Load string length  
   LOOP CKSM 1,2 Compute checksum  
   BNZ LOOP Repeat if not done  
   SR 0,0 Zero for divide  
   D 0,SIZE Compute hash value  

4. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.  

5. The storage-operand references of CHECKSUM may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98)  

6. Figure 7-2 on page 7-34 contains a summary of the operation.
Figure 7-2. Execution of CHECKSUM
CIPHER MESSAGE (KM)

KM  R₁, R₂  [RRE]

'B92E'  ///////  R₁  R₂

0  16  24  28  31

CIPHER MESSAGE WITH CHAINING (KMC)

KMC  R₁, R₂  [RRE]

'B92F'  ///////  R₁  R₂

0  16  24  28  31

A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figures 7-3 and 7-4 show the assigned function codes for CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING, respectively. All other function codes are unassigned. For cipher functions, bit 56 is the modifier bit which specifies whether an encryption or a decryption operation is to be performed. The modifier bit is ignored for all other functions. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for CIPHER MESSAGE are as follows.

---

**Figure 7-3. Function Codes for CIPHER MESSAGE**

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
<th>Parm. Block Size (bytes)</th>
<th>Data Block Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KM-Query</td>
<td>16</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>KM-DEA</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>KM-TDEA-128</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>KM-TDEA-192</td>
<td>24</td>
<td>8</td>
</tr>
</tbody>
</table>

**Explanation:**

— Not applicable

---

The function codes for CIPHER MESSAGE WITH CHAINING are as follows.

**Figure 7-4. Function Codes for CIPHER MESSAGE WITH CHAINING**

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
<th>Parm. Block Size (bytes)</th>
<th>Data Block Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KMC-Query</td>
<td>16</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>KMC-DEA</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>KMC-TDEA-128</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>KMC-TDEA-192</td>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

**Explanation:**

— Not applicable

---

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R₁, R₂, and R₁ + 1 are ignored for the query function.

For all other functions, the second operand is ciphered as specified by the function code using a cryptographic key in the parameter block, and the result is placed in the first-operand location. For CIPHER MESSAGE WITH CHAINING, ciphering also uses an initial chaining value in the parameter block, and the chaining value is updated as part of the operation.
The \( R_1 \) field designates a general register and must designate an even-numbered register; otherwise, a specification exception is recognized.

The \( R_2 \) field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first and second operands is specified by the contents of the \( R_1 \) and \( R_2 \) general registers, respectively. The number of bytes in the second-operand location is specified in general register \( R_2 + 1 \). The first operand is the same length as the second operand.

As part of the operation, the addresses in general registers \( R_1 \) and \( R_2 \) are incremented by the number of bytes processed, and the length in general register \( R_2 + 1 \) is decremented by the same number. The formation and updating of the addresses and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers \( R_1 \) and \( R_2 \) constitute the addresses of the first and second operands, respectively, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated addresses replace the corresponding bits in general registers \( R_1 \) and \( R_2 \), carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general registers \( R_1 \) and \( R_2 \) is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general registers \( R_1 \) and \( R_2 \) constitute the addresses of the first and second operands, respectively; bits 0-63 of the updated addresses replace the contents of general registers \( R_1 \) and \( R_2 \), and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register \( R_2 + 1 \) form a 32-bit unsigned binary integer which specifies the number of bytes in the first and second operands, and the contents of bit positions 0-31 are ignored; bits 32-63 of the updated value replace the corresponding bits in general register \( R_2 + 1 \). In the 64-bit addressing mode, the contents of bit positions 0-63 of general register \( R_2 + 1 \) form a 64-bit unsigned binary integer which specifies the number of bytes in the first and second operands; and the updated value replaces the contents of general register \( R_2 + 1 \).

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers \( R_1 \), \( R_2 \), and \( R_2 + 1 \), always remain unchanged.

**Figure 7-5 on page 7-37** shows the contents of the general registers just described.
In the access-register mode, access registers 1, \( R_1 \), and \( R_2 \) specify the address spaces containing the parameter block, first, and second operands, respectively. The result is obtained as if processing starts at the left end of both the first and second operands and proceeds to the right, block by block. The operation is ended when the number of bytes in the second operand as specified in general register...
R₂ + 1 have been processed and placed at the first-operand location (called normal completion) or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

The results in the first-operand location and the chaining-value field are unpredictable if any of the following situations occur:

1. The cryptographic-key field overlaps any portion of the first operand.
2. The chaining-value field overlaps any portion of the first operand or the second operand.
3. The first and second operands overlap destructively. Operands are said to overlap destructively when the first-operand location would be used as a source after data would have been moved into it, assuming processing to be performed from left to right and one byte at a time.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in R₂ + 1 is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in R₂ + 1 is nonzero.

When a storage-alteration PER event is recognized, fewer than 4K additional bytes are stored into the first-operand locations before the event is reported.

When the second-operand length is initially zero, the parameter block, first, and second operands are not accessed, general registers R₁, R₂, and R₂ + 1 are not changed, and condition code 0 is set.

When the contents of the R₁ and R₂ fields are the same, the contents of the designated registers are incremented only by the number of bytes processed, not by twice the number of bytes processed.

As observed by other CPUs and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

In certain unusual situations, instruction execution may complete by setting condition code 3 without updating the registers and chaining value to reflect the last unit of the first and second operands processed. The size of the unit processed in this case depends on the situation and the model, but is limited such that the portion of the first and second operands which have been processed and not reported do not overlap in storage. In all cases, change bits are set and PER storage-alteration events are reported, when applicable, for all first-operand locations processed.

Access exceptions may be reported for a larger portion of an operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of an operand nor for locations more than 4K bytes beyond the current location being processed.

Symbols Used in Function Descriptions

The following symbols are used in the subsequent description of the CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key.

Further description of the data-encryption algorithm may be found in Data Encryption Algorithm, ANSI-X3.92.1981, American National Standard for Information Systems.
KM-Query (KM Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
  0
  8
63
```

Figure 7-8. Parameter Block for KM-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KM-Query function completes; condition code 3 is not applicable to this function.

KM-DEA (KM Function Code 1)

The parameter block used for the function has the following format:

```
<table>
<thead>
<tr>
<th>0</th>
<th>Cryptographic Key (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 7-9. Parameter Block for KM-DEA

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in the following figure:

```
Parameter Block
in Storage
K <8>

Op 2 in Storage
P1 <8> P2 <8> P3 <8>  Pn <8>

K ─/SM59≤≤≤≤│DEA│ K ─/SM59≤≤≤≤│DEA│ K ─/SM59≤≤≤≤│DEA│ K ─/SM59≤≤≤≤│DEA│
│ e │ │ e │ │ e │ │ e │
└─┬─┘ └─┬─┘ └─┬─┘ └─┬─┘
```

Figure 7-10. KM-DEA Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The operation is shown in the following figure:
**KM-TDEA-128 (KM Function Code 2)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cryptographic Key 1 (K1)</td>
<td>Cryptographic Key 2 (K2)</td>
</tr>
</tbody>
</table>

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA (triple DEA) algorithm with the two 64-bit cryptographic keys in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in the following figure:

**Figure 7-14. KM-TDEA-128 Encipher Operation**

**KM-TDEA-192 (KM Function Code 3)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.
The parameter block used for the function has the following format:

<table>
<thead>
<tr>
<th>0</th>
<th>Cryptographic Key 1 (K1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Cryptographic Key 2 (K2)</td>
</tr>
<tr>
<td>16</td>
<td>Cryptographic Key 3 (K3)</td>
</tr>
</tbody>
</table>

Figure 7-15. Parameter Block for KM-TDEA-192

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys in the parameter block. Each plaintext block is independently enciphered; that is, the encipher operation is performed without chaining. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The operation is shown in the following figure:

Figure 7-16. KM-TDEA-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys in the parameter block. Each ciphertext block is independently deciphered; that is, the decipher operation is performed without chaining. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The operation is shown in the following figure:

Figure 7-17. KM-TDEA-192 Decipher Operation

KMC-Query (KMC Function Code 0)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

0
8

Status Word

Figure 7-18. Parameter Block for KMC-Query

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the CIPHER MESSAGE WITH CHAINING instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KMC-Query function completes; condition code 3 is not applicable to this function.

KMC-DEA (KMC Function Code 1)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:
When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in the following figure:

```
Figure 7-20. KMC-DEA Encipher Operation
```

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in the following figure:

```
Figure 7-21. KMC-DEA Decipher Operation
```

KMC-TDEA-128 (KMC Function Code 2)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

```
Figure 7-22. Parameter Block for KMC-TDEA-128
```

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in the following figure:
ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in the following figure:

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 2. The last ciphertext block is the output plaintext block. The operation is shown in the following figure.

KMC-TDEA-192 (KMC Function Code 3)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-5 on page 7-37.

The parameter block used for the function has the following format:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Chaining Value (CV)</td>
</tr>
<tr>
<td>8</td>
<td>Cryptographic Key 1 (K1)</td>
</tr>
<tr>
<td>16</td>
<td>Cryptographic Key 2 (K2)</td>
</tr>
<tr>
<td>24</td>
<td>Cryptographic Key 3 (K3)</td>
</tr>
</tbody>
</table>

When the modifier bit in general register 0 is zero, an encipher operation is performed. The 8-byte plaintext blocks (P1, P2, ..., Pn) in operand 2 are enciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first ciphertext block is the chaining value in the parameter block; the chaining value for deriving each subsequent ciphertext block is the corresponding previous
ciphertext block. The ciphertext blocks (C1, C2, ..., Cn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field of the parameter block. The operation is shown in the following figure:

Figure 7-26. KMC-TDEA-192 Encipher Operation

When the modifier bit in general register 0 is one, a decipher operation is performed. The 8-byte ciphertext blocks (C1, C2, ..., Cn) in operand 2 are deciphered using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The chaining value, called the initial chaining value (ICV), for deriving the first plaintext block is in the parameter block; the chaining value for deriving each subsequent plaintext block is the corresponding previous ciphertext block. The plaintext blocks (P1, P2, ..., Pn) are stored in operand 1. The last ciphertext block is the output chaining value (OCV) and is stored into the chaining-value field in the parameter block. The operation is shown in the following figure:

Figure 7-27. KMC-TDEA-192 Decipher Operation

Special Conditions for KM and KMC

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
2. The R₁ or R₂ field designates an odd-numbered register or general register 0.
3. The second operand length is not a multiple of the data block size of the designated function (see Figure 7-3 on page 7-35 to determine the data block sizes for CIPHER MESSAGE functions; see Figure 7-4 on page 7-35 to determine the data block sizes for CIPHER MESSAGE WITH CHAINING functions). This specification-exception condition does not apply to the query functions.

Resulting Condition Code:

0  Normal completion
1  --
2  --
3  Partial completion

Program Exceptions:

- Access (fetch, operand 2 and cryptographic key; store, operand 1; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)
- Specification

1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Operation exception.

8. Specification exception due to invalid function code or invalid register number.

9. Specification exception due to invalid operand length.

10. Condition code 0 due to second-operand length originally zero.

11. Access exceptions for an access to the parameter block, first, or second operand.

12. Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).

13. Condition code 3 due to partial completion (second-operand length still nonzero).

Figure 7-28. Priority of Execution: KM and KMC

Programming Notes:

1. When condition code 3 is set, the general registers containing the operand addresses and length, and, for CIPHER MESSAGE WITH CHAINING, the chaining value in the parameter block, are usually updated such that the program can simply branch back to the instruction to continue the operation.

For unusual situations, the CPU protects against endless reoccurrence of the non-progress case and also protects against setting condition code 3 when the portion of the first and second operands to be reprocessed overlap in storage. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop and no exposure to incorrectly retrying the instruction.

2. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3. For CIPHER MESSAGE WITH CHAINING, the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.

3. To save storage, the first and second operands may overlap exactly or the starting point of the first operand may be to the left of the starting point of the second operand. In either case, the overlap is not destructive.

COMPARE

CR R1,R2 [RR]

\[
\begin{array}{ccc}
'19' & R1 & R2 \\
0 & 8 & 12 & 15 \\
\end{array}
\]

CGR R1,R2 [RRE]

\[
\begin{array}{cccc}
'B920' & \overset{\text{////////}}{\_\_\_\_\_\_\_\_} & R1 & R2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]

CGFR R1,R2 [RRE]

\[
\begin{array}{cccc}
'B930' & \overset{\text{////////}}{\_\_\_\_\_\_\_\_} & R1 & R2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]
The first operand is compared with the second operand, and the result is indicated in the condition code. For COMPARE (CR, C, CY), the operands are treated as 32-bit signed binary integers. For COMPARE (CGR, CG), they are treated as 64-bit signed binary integers. For COMPARE (CGFR, CGF), the second operand is treated as a 32-bit signed binary integer, and the first operand is treated as a 64-bit signed binary integer.

The displacement for C is treated as a 12-bit unsigned binary integer. The displacement for CY, CG, and CGF is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

0  Operands equal  
1  First operand low  
2  First operand high  
3  --

**Program Exceptions:**

- Access (fetch, operand 2 of C, CY, CG, and CGF only)
- Operation (CY, if the long-displacement facility is not installed)

The second-operand address is not used to address data. Bits 49-62 of the second-operand address, with one rightmost and one leftmost zero appended, are used as a 16-bit index limit. Bit 63 of the second-operand address is the operand-control bit. When bit 63 is zero, the codeword is formed from the high operand; when bit 63 is one, the codeword is formed from the low operand. The remainder of the second-operand address is ignored.

General registers 1 and 3 contain the base addresses of the first and third operands. Bits 48-63 of general register 2 are used as an index for addressing both the first and third operands. General registers 1, 2, and 3 must all initially contain even values; otherwise, a specification exception is recognized.

In the access-register mode, access register 1 specifies the address space containing the first and third operands.

The size of the units by which the first and third operands are compared, the size of the resulting codeword, and the participation of bits 0-31 of general registers 1, 2, and 3 in the operation depend on the addressing mode. In the 24-bit or 31-bit addressing mode, the comparison unit is two bytes, the codeword is four bytes, and bits 0-31 are ignored and remain unchanged. In the 64-bit addressing mode, the comparison unit is six bytes, the codeword is eight bytes, and bits 0-31 are used in and may be changed by the operation.
The operation consists in comparing the first and third operands halfword by halfword and incrementing the index until an unequal pair of halfwords is found or the index exceeds the index limit. This proceeds in units of operation, between which interruptions may occur.

At the start of a unit of operation, the index, bits 48-63 of general register 2, is logically compared with the index limit. If the index is larger, the instruction is completed by placing bits 32-63 of general register 3, with bit 32 set to one, in bit positions 32-63 of general register 2, and by setting condition code 0.

If the index is less than or equal to the index limit, the index is applied to the first-operand and third-operand base addresses to locate the current pair of halfwords to be compared. The index, with 48 leftmost zeros appended, and bits 32-63 of general register 1, with 32 leftmost zeros appended, are added to form a 64-bit intermediate value. A carry out of bit position 32, if any, is ignored. The address of the current first-operand halfword is generated from the intermediate value by following the normal rules for operand address generation. The address of the current third-operand halfword is formed in the same manner by adding bits 32-63 of general register 3 and the index.

The current first-operand and third-operand halfwords are logically compared. If they are equal, the contents of general register 2 are incremented by 2, and a unit of operation ends.

If the compare values are unequal, the contents of general register 2 are incremented by 2 and then shifted left logically by 16 bit positions. The shifting occurs only within bit positions 32-63. If the operand-control bit is zero, (1) the one's complement of the higher halfword is placed in bit positions 48-63 of general register 2, and (2) if operand 1 was higher, bits 32-63 of general registers 1 and 3 are interchanged. If the operand-control bit is one, (1) the lower halfword is placed in bit positions 48-63 of general register 2, and (2) if operand 1 was lower, bits 32-63 of general registers 1 and 3 are interchanged.

For the purpose of recognizing access exceptions, operand 1 and operand 3 are both considered to have a length equal to 2 more than the value of the index limit minus the index.

**Operation in the 64-Bit Addressing Mode**

The operation consists in comparing the first and third operands in units of six bytes at a time and incrementing the index until an unequal pair of six-byte units is found or the index exceeds the index limit. This proceeds in units of operation, between which interruptions may occur.

At the start of a unit of operation, the index, bits 48-63 of general register 2, is logically compared with the index limit. If the index is larger, the instruction is completed by placing bits 0-63 of general register 3, with bit 0 set to one, in bit positions 0-63 of general register 2, and by setting condition code 0.

If the index is less than or equal to the index limit, the index is applied to the first-operand and third-operand base addresses to locate the current pair of six-byte units to be compared. The index, with 48 leftmost zeros appended, and bits 0-63 of general register 1 are added to form the 64-bit address of the current first-operand six-byte unit. A carry out of bit position 0, if any, is ignored. The address of the current third-operand six-byte unit is formed in the same manner by adding bits 0-63 of general register 3 and the index.

The current first-operand and third-operand six-byte units are logically compared. If they are equal, the contents of general register 2 are incremented by 6, and a unit of operation ends.

If the compare values are unequal, the contents of general register 2 are incremented by 6 and then shifted left logically by 48 bit positions. If the operand-control bit is zero, (1) the one's complement of the higher six-byte unit is placed in bit positions 16-63 of general register 2, and (2) if operand 1 was higher, bits 0-63 of general registers 1 and 3 are interchanged. If the operand-control bit is one, (1) the lower six-byte unit is placed in bit positions 16-63 of general register 2, and (2) if operand 1 was lower, bits 0-63 of general registers 1 and 3 are interchanged.

For the purpose of recognizing access exceptions, operand 1 and operand 3 are both considered to have a length equal to 6 more than the value of the index limit minus the index.
Specifications Independent of Addressing Mode

The condition code is unpredictable if the instruction is interrupted.

When the index is initially larger than the index limit, access exceptions are not recognized for the storage operands. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the byte being processed. Access exceptions are not recognized when a specification-exception condition exists.

If the B2 field designates general register 2, it is unpredictable whether or not the index limit is recomputed; thus, in this case the operand length is unpredictable. However, in no case can the operands exceed $2^{15}$ bytes in length.

Resulting Condition Code:

0  Operands equal
1  Operand-control bit zero and operand 1 low, or operand-control bit one and operand 3 low
2  Operand-control bit zero and operand 1 high, or operand-control bit one and operand 3 high
3  --

Program Exceptions:

- Access (fetch, operands 1 and 3)
- Specification

Programming Notes:

1. An example of the use of COMPARE AND FORM CODEWORD is given in "Sorting Instructions" in Appendix A, "Number Representation and Instruction-Use Examples."

2. The offset of the halfword or six-byte unit (depending on the addressing mode) of the first and third operands at which comparison is to begin should be placed in bit positions 48-63 of general register 2 before executing COMPARE AND FORM CODEWORD. The index limit derived from the second-operand address should be the offset of the last halfword or six-byte unit of the first and third operands for which comparison can be made. When the operands do not compare equal, the leftmost 16 bits of the codeword formed in general register 2 (bits 32-47 of the register in the 24-bit or 31-bit addressing mode, or bits 0-15 in the 64-bit addressing mode) by the execution of COMPARE AND FORM CODEWORD gives the offset of the first halfword or six-byte unit not compared. If the codewords compare equal in an UPDATE TREE operation, bit positions 32-47 of general register 2 in the 24-bit or 31-bit addressing mode, or bit positions 0-15 in the 64-bit addressing mode, will contain the offset at which another COMPARE AND FORM CODEWORD should resume comparison for breaking codeword ties. Operand-control-bit values of zero or one are used for sorting operands in ascending or descending order, respectively. Refer to "Sorting Instructions" on page A-51 for a discussion of the use of codewords in sorting.

3. The condition code indicates the results of comparing operands up to 32,768 bytes long. Equal operands result in a negative codeword in bit positions 32-63 of general register 2 in the 24-bit or 31-bit addressing mode, or in bit positions 0-63 in the 64-bit addressing mode. A negative codeword also results in the 24-bit or 31-bit mode when the index limit is 32,766 and the operands that are compared differ in only their last two bytes, or in the 64-bit mode when the limit is 32,762 and the operands differ in only their last six bytes. If this latter codeword is used by UPDATE TREE, an incorrect result may be indicated in general registers 0 and 1. Therefore, the index limit should not exceed 32,764 in the 24-bit or 31-bit mode, or 32,760 in the 64-bit mode, when the resulting codeword is to be used by UPDATE TREE.

4. Special precautions should be taken if COMPARE AND FORM CODEWORD is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

5. Further programming notes concerning interruptible instructions are included in "Interruptible Instructions" in Chapter 5, "Program Execution."

6. The storage-operand references of COMPARE AND FORM CODEWORD may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98.)

7. Figure 7-29 on page 7-49 and Figure 7-30 on page 7-50 contain summaries of the operation in the 24-bit or 31-bit addressing mode,
and Figure 7-31 on page 7-51 and Figure 7-32 on page 7-52 contain summaries of the operation in the 64-bit addressing mode.

<table>
<thead>
<tr>
<th>Operand-Control Bit</th>
<th>Relation</th>
<th>Resulting Condition Code</th>
<th>Result in GR2 (Bits 32-63)</th>
<th>Result in GR1 (Bits 32-63)</th>
<th>Result in GR3 (Bits 32-63)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>op1 = op3</td>
<td>0</td>
<td>OGR3b1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>op1 &lt; op3</td>
<td>1</td>
<td>X, nop3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>op1 &gt; op3</td>
<td>2</td>
<td>X, nop1 OGR3 OGR1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>op1 = op3</td>
<td>0</td>
<td>OGR3b1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>op1 &lt; op3</td>
<td>2</td>
<td>X, top1 OGR3 OGR1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>op1 &gt; op3</td>
<td>1</td>
<td>X, top3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation:**
- The bits remain unchanged.
- OGR1 The original value of GR1 bits 32-63.
- OGR3 The original value of GR3 bits 32-63.
- OGR3b1 The original value of GR3 bits 32-63 with bit 32 set to one.
- X Bits 32-47 of GR2 are 2 more than the index of the first unequal halfword.
- nop1 Bits 48-63 of GR2 are the one's complement of the first unequal halfword in operand 1.
- nop3 Bits 48-63 of GR2 are the one's complement of the first unequal halfword in operand 3.
- top1 Bits 48-63 of GR2 are the first unequal halfword in operand 1.
- top3 Bits 48-63 of GR2 are the first unequal halfword in operand 3.

*Figure 7-29. Operation of COMPARE AND FORM CODEWORD in the 24-Bit or 31-bit Addressing Mode*
Figure 7-30. Execution of COMPARE AND FORM CODEWORD in the 24-Bit or 31-bit Addressing Mode
<table>
<thead>
<tr>
<th>Operand-Control Bit</th>
<th>Relation</th>
<th>Resulting Condition</th>
<th>Result in GR2 (Bits 0-63)</th>
<th>Result in GR1 (Bits 0-63)</th>
<th>Result in GR3 (Bits 0-63)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>op1 = op3</td>
<td>0</td>
<td>OGR3b1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>op1 &lt; op3</td>
<td>1</td>
<td>X, nop3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>op1 &gt; op3</td>
<td>2</td>
<td>X, nop1</td>
<td>OGR3</td>
<td>OGR1</td>
</tr>
<tr>
<td>1</td>
<td>op1 = op3</td>
<td>0</td>
<td>OGR3b1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>op1 &lt; op3</td>
<td>2</td>
<td>X, top1</td>
<td>OGR3</td>
<td>OGR1</td>
</tr>
<tr>
<td>1</td>
<td>op1 &gt; op3</td>
<td>1</td>
<td>X, top3</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation:**
- The bits remain unchanged.
- OGR1 The original value of GR1 bits 0-63.
- OGR3 The original value of GR3 bits 0-63.
- OGR3b1 The original value of GR3 bits 0-63 with bit 0 set to one.
- X Bits 0-15 of GR2 are 6 more than the index of the first unequal six-byte unit.
- nop1 Bits 16-63 of GR2 are the one's complement of the first unequal six-byte unit in operand 1.
- nop3 Bits 16-63 of GR2 are the one's complement of the first unequal six-byte unit in operand 3.
- top1 Bits 16-63 of GR2 are the first unequal six-byte unit in operand 1.
- top3 Bits 16-63 of GR2 are the first unequal six-byte unit in operand 3.

*Figure 7-31. Operation of COMPARE AND FORM CODEWORD in the 64-Bit Addressing Mode*
Figure 7-32. Execution of COMPARE AND FORM CODEWORD in the 64-Bit Addressing Mode
COMPARE AND SWAP

CS \( R_1, R_3, D_2(B_2) \) \([RS]\)

\[
\begin{array}{cccc}
'BA' & R_1 & R_3 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 31
\end{array}
\]

CSY \( R_1, R_3, D_2(B_2) \) \([RSY]\)

\[
\begin{array}{ccccc}
'EB' & R_1 & R_3 & B_2 & DL_2 & DH_2 & '14' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47
\end{array}
\]

CSG \( R_1, R_3, D_2(B_2) \) \([RSY]\)

\[
\begin{array}{ccccc}
'EB' & R_1 & R_3 & B_2 & DL_2 & DH_2 & '30' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47
\end{array}
\]

COMPARE DOUBLE AND SWAP

CDS \( R_1, R_3, D_2(B_2) \) \([RS]\)

\[
\begin{array}{cccc}
'BB' & R_1 & R_3 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 31
\end{array}
\]

CDSY \( R_1, R_3, D_2(B_2) \) \([RSY]\)

\[
\begin{array}{ccccc}
'EB' & R_1 & R_3 & B_2 & DL_2 & DH_2 & '31' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47
\end{array}
\]

CDSG \( R_1, R_3, D_2(B_2) \) \([RSY]\)

\[
\begin{array}{ccccc}
'EB' & R_1 & R_3 & B_2 & DL_2 & DH_2 & '3E' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47
\end{array}
\]

The first and second operands are compared. If they are equal, the third operand is stored at the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. The result of the comparison is indicated in the condition code.

For COMPARE AND SWAP (CS, CSY), the first and third operands are 32 bits in length, with each operand occupying bit positions 0-63 of a general register. The second operand is a word in storage.

For COMPARE AND SWAP (CSG), the first and third operands are 64 bits in length, with each operand occupying bit positions 0-63 of the even-numbered register of an even-odd pair of general registers, and the second 32 bits occupy bit positions 0-31 of the odd-numbered register of the pair. The second operand is a doubleword in storage.

For COMPARE DOUBLE AND SWAP (CDS, CDSY), the first and third operands are 64 bits in length. The first 32 bits of an operand occupy bit positions 32-63 of the even-numbered register of an even-odd pair of general registers, and the second 32 bits occupy bit positions 32-63 of the odd-numbered register of the pair. The second operand is a doubleword in storage.

When an equal comparison occurs, the third operand is stored at the second-operand location. The fetch of the second operand for purposes of comparison and the store into the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs.

When the result of the comparison is unequal, the second operand is loaded at the first-operand location, and the second-operand location remains unchanged. However, on some models, the contents may be fetched and subsequently stored back unchanged at the second-operand location. This update appears to be a block-concurrent interlocked-update reference as observed by other CPUs.

A serialization function is performed before the operand is fetched and again after the operation is completed.

The displacement for CS and CDS is treated as a 12-bit unsigned binary integer. The displacement for CSY, CSG, CDSY, and CDSG is treated as a 20-bit signed binary integer.

The second operand of COMPARE AND SWAP (CS, CSY) must be designated on a word boundary. The second operand of COMPARE
AND SWAP (CSG) and COMPARE DOUBLE AND SWAP (CDS, CDSY) must be designated on a doubleword boundary. The second operand of COMPARE DOUBLE AND SWAP (CDSG) must be designated on a quadword boundary. The R₁ and R₃ fields for COMPARE DOUBLE AND SWAP must each designate an even-numbered register. Otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0 First and second operands equal, second operand replaced by third operand
1 First and second operands unequal, first operand replaced by second operand
2 --
3 --

**Program Exceptions:**

- Access (fetch and store, operand 2)
- Operation (CSY and CDSY, if the long-displacement facility is not installed)
- Specification

**Programming Notes:**

1. Several examples of the use of the COMPARE AND SWAP and COMPARE DOUBLE AND SWAP instructions are given in Appendix A, "Number Representation and Instruction-Use Examples."

2. Some of the following notes are worded, with respect to operand size, for CS, CSY, CDS, and CDSY. Similar notes, worded for a larger operand size, would apply to CSG and CDSG.

3. COMPARE AND SWAP can be used by CPU programs sharing common storage areas in either a multiprogramming or multiprocessing environment. Two examples are:

   a. By performing the following procedure, a CPU program can modify the contents of a storage location even though the possibility exists that the CPU program may be interrupted by another CPU program that will update the location or that another CPU program may simultaneously update the location. First, the entire word containing the byte or bytes to be updated is loaded into a general register. Next, the updated value is computed and placed in another general register. Then COMPARE AND SWAP is executed with the R₁ field designating the register that contains the original value and the R₃ field designating the register that contains the updated value. If the update has been successful, condition code 0 is set. If the storage location no longer contains the original value, the update has not been successful, the general register designated by the R₁ field of the COMPARE AND SWAP instruction contains the new current value of the storage location, and condition code 1 is set. When condition code 1 is set, the CPU program can repeat the procedure using the new current value.

   b. COMPARE AND SWAP can be used for controlled sharing of a common storage area, including the capability of leaving a message (in a chained list of messages) when the common area is in use. To accomplish this, a word in storage can be used as a control word, with a zero value in the word indicating that the common area is not in use and that no messages exist, a negative value indicating that the area is in use and that no messages exist, and a nonzero positive value indicating that the common area is in use and that the value is the address of the most recent message added to the list. Thus, any number of CPU programs desiring to seize the area can use COMPARE AND SWAP to update the control word to indicate that the area is in use or to add messages to the list. The single CPU program which has seized the area can also safely use COMPARE AND SWAP to remove messages from the list.

4. COMPARE DOUBLE AND SWAP can be used in a manner similar to that described for COMPARE AND SWAP. In addition, it has another use. Consider a chained list, with a control word used to address the first message in the list, as described in programming note 3b above. If multiple CPU programs are to be permitted to delete messages by using COMPARE AND SWAP (and not just the single CPU program which has seized the common area), there is a possibility the list will be incorrectly updated. This would occur if, for example, after one CPU program has fetched the address of the most recent message in order to remove the message,
another CPU program removes the first two messages and then adds the first message back into the chain. The first CPU program, on continuing, cannot easily detect that the list is changed. By increasing the size of the control word to a doubleword containing both the first message address and a word with a change number that is incremented for each modification of the list, and by using COMPARE DOUBLE AND SWAP to update both fields together, the possibility of the list being incorrectly updated is reduced to a negligible level. That is, an incorrect update can occur only if the first CPU program is delayed while changes exactly equal in number to a multiple of \(2^n\) take place and only if the last change places the original message address in the control word.

5. COMPARE AND SWAP and COMPARE DOUBLE AND SWAP do not interlock against storage accesses by channel programs. Therefore, the instructions should not be used to update a location at which a channel program may store, since the channel-program data may be lost.

6. To ensure successful updating of a common storage field by two or more CPUs, all updates must be done by means of an interlocked-update reference. COMPARE AND SWAP, COMPARE AND SWAP AND PURGE, COMPARE DOUBLE AND SWAP, and TEST AND SET are the only instructions that perform an interlocked-update reference. For example, if one CPU executes OR IMMEDIATE and another CPU executes COMPARE AND SWAP to update the same byte, the fetch by OR IMMEDIATE may occur either before the fetch by COMPARE AND SWAP or between the fetch and the store by COMPARE AND SWAP, and then the store by OR IMMEDIATE may occur after the store by COMPARE AND SWAP, in which case the change made by COMPARE AND SWAP is lost.

7. For the case of a condition-code setting of 1, COMPARE AND SWAP and COMPARE DOUBLE AND SWAP may or may not, depending on the model, cause any of the following to occur for the second-operand location: a PER storage-alteration event may be recognized; a protection exception for storing may be recognized; and, provided no access exception exists, the change bit may be set to one. Because the contents of storage remain unchanged, the change bit may or may not be one when a PER storage-alteration event is recognized.

8. The performance of CDSG on some models may be significantly slower than that of CSG. When quadword consistency is not required by the program, alternate code sequences should be used.

### COMPARE HALFWORD

**CH**  
R<sub>1</sub>, D<sub>2</sub>(X<sub>2</sub>, B<sub>2</sub>) [RX]

<table>
<thead>
<tr>
<th>'49'</th>
<th>R&lt;sub&gt;1&lt;/sub&gt;</th>
<th>X&lt;sub&gt;2&lt;/sub&gt;</th>
<th>B&lt;sub&gt;2&lt;/sub&gt;</th>
<th>D&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 12 16 20 31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CHY**  
R<sub>1</sub>, D<sub>2</sub>(X<sub>2</sub>, B<sub>2</sub>) [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R&lt;sub&gt;1&lt;/sub&gt;</th>
<th>X&lt;sub&gt;2&lt;/sub&gt;</th>
<th>B&lt;sub&gt;2&lt;/sub&gt;</th>
<th>DL&lt;sub&gt;2&lt;/sub&gt;</th>
<th>DH&lt;sub&gt;2&lt;/sub&gt;</th>
<th>'79'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 12 16 20 32 40 47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### COMPARE HALFWORD IMMEDIATE

**CHI**  
R<sub>1</sub>, I<sub>2</sub> [RI]

<table>
<thead>
<tr>
<th>'A7'</th>
<th>R&lt;sub&gt;1&lt;/sub&gt;</th>
<th>'E'</th>
<th>I&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 12 16 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CGHI**  
R<sub>1</sub>, I<sub>2</sub> [RI]

<table>
<thead>
<tr>
<th>'A7'</th>
<th>R&lt;sub&gt;1&lt;/sub&gt;</th>
<th>'F'</th>
<th>I&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 12 16 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first operand is compared with the second operand, and the result is indicated in the condition code. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For COMPARE HALFWORD (CH, CHY) and COMPARE HALFWORD IMMEDIATE (CHI), the first operand is treated as a 32-bit signed binary integer. For COMPARE HALFWORD IMMEDIATE (CGHI), the first operand is treated as a 64-bit signed binary integer.
The displacement for CH is treated as a 12-bit unsigned binary integer. The displacement for CHY is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

0  Operands equal  
1  First operand low  
2  First operand high  
3  --

**Program Exceptions:**

- Access (fetch, operand 2 of CH, CHY only)
- Operation (CHY, if the long-displacement facility is not installed)

**Programming Note:** An example of the use of the COMPARE HALFWORD instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

**COMPARE LOGICAL**

CLR  R₁,R₂  [RR]

<table>
<thead>
<tr>
<th>'15'</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

CLGR  R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>'B921'</th>
<th>/ / / / / /</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

CLGFR  R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>'B931'</th>
<th>/ / / / / /</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

CL  R₁,D₂(X₂,B₂)  [RX]

<table>
<thead>
<tr>
<th>'55'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

CLY  R₁,D₂(X₂,B₂)  [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'55'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The first operand is compared with the second operand, and the result is indicated in the condition code.

For COMPARE LOGICAL (CLR, CL, CLY), the operands are treated as 32 bits. For COMPARE LOGICAL (CLGR, CLG), the operands are treated as 64 bits. For COMPARE LOGICAL (CLGFR, CLGF), the first operand is treated as 64 bits, and the second operand is treated as 32 bits with 32 zeros appended on the left.

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the fields is reached. For COMPARE LOGICAL (CL, CLY, CLG, CLGF, CLC), access exceptions may or may not be recognized for the portion of a storage operand to the right of the first unequal byte.

The displacements for CL, CLI, and both operands of CLC are treated as 12-bit unsigned binary integers. The displacement for CLY, CLG, CLGF,
and CLIY is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**
0  Operands equal
1  First operand low
2  First operand high
3  --

**Program Exceptions:**
- Access (fetch, operand 2, CL, CLY, CLG, CLGF, and CLC; fetch, operand 1, CLI, CLIY, and CLC)
- Operation (CLY and CLIY if the long-displacement facility is not installed)

**Programming Notes:**
1. Examples of the use of the COMPARE LOGICAL instruction are given in Appendix A, "Number Representation and Instruction-Use Examples."
2. COMPARE LOGICAL treats all bits of each operand alike as part of a field of unstructured logical data. For COMPARE LOGICAL (CLC), the comparison may extend to field lengths of 256 bytes.

**COMPARE LOGICAL CHARACTERS UNDER MASK**

<table>
<thead>
<tr>
<th>CLM</th>
<th>R1, M3, D2 (B2) [RS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'BD'</td>
<td>R1 12 16 20 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLMY</th>
<th>R1, M3, D2 (B2) [RSY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EB'</td>
<td>R1 12 16 20 DL2 DH2 '21'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CLMH</th>
<th>R1, M3, D2 (B2) [RSY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EB'</td>
<td>R1 12 16 20 DL2 DH2 '20'</td>
</tr>
</tbody>
</table>

The first operand is compared with the second operand under control of a mask, and the result is indicated in the condition code.

The contents of the M3 field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register R1. For COMPARE LOGICAL CHARACTERS UNDER MASK (CLM, CLMY), the four bytes to which the mask bits correspond are in bit positions 32-63 of general register R1. For COMPARE LOGICAL CHARACTERS UNDER MASK (CLMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The byte positions corresponding to ones in the mask are considered as a contiguous field and are compared with the second operand. The second operand is a contiguous field in storage, starting at the second-operand address and equal in length to the number of ones in the mask. The bytes in the general register corresponding to zeros in the mask do not participate in the operation.

The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the fields is reached.

When the mask is not zero, exceptions associated with storage-operand access are recognized for no more than the number of bytes specified by the mask. Access exceptions may or may not be recognized for the portion of a storage operand to the right of the first unequal byte. When the mask is zero, access exceptions are recognized for one byte at the second-operand address.

The displacement for CLM is treated as a 12-bit unsigned binary integer. The displacement for CLMY and CLMH is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**
0  Operands equal, or mask bits all zeros
1  First operand low
2  First operand high
3  --

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation (CLMY, if the long-displacement facility is not installed)

**Programming Note:** An example of the use of the COMPARE LOGICAL CHARACTERS UNDER MASK instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
COMPARE LOGICAL LONG

CLCL R₁, R₂ [RR]

<table>
<thead>
<tr>
<th>'0F'</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

The first operand is compared with the second operand, and the result is indicated in the condition code. The shorter operand is considered to be extended on the right with padding bytes.

The R₁ and R₂ fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R₁ and R₂, respectively. The number of bytes in the first-operand and second-operand locations is specified by unsigned binary integers in bit positions 40-63 of general registers R₁ + 1 and R₂ + 1, respectively. Bit positions 32-39 of general register R₂ + 1 contain the padding byte. The contents of bit positions 0-39 of general register R₁ + 1 and of bit positions 0-31 of general register R₂ + 1 are ignored.

The handling of the addresses in general registers R₁ and R₂ is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R₁ and R₂ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-33 on page 7-59.
The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found or the end of the longer operand is reached. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If both operands are of zero length, the operands are considered to be equal.

Figure 7-33. Register Contents for COMPARE LOGICAL LONG

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the lengths in general registers \( R_1 + 1 \) and \( R_2 + 1 \) are decremented by the number of bytes compared, and the addresses in general registers \( R_1 \) and \( R_2 \) are incremented by the same number, so that the instruction, when reexecuted, resumes at the point of interruption. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers \( R_1 \) and \( R_2 \) are set to zeros, and the contents of bit positions 0-31
remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged, and the condition code is unpredictable. If the operation is interrupted after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and its address is updated accordingly.

If the operation ends because of an inequality, the address fields in general registers $R_1$ and $R_2$ at completion identify the first unequal byte in each operand. The lengths in bit positions 40-63 of general registers $R_1 + 1$ and $R_2 + 1$ are decremented by the number of bytes that were equal, unless the inequality occurred with the padding byte, in which case the length field for the shorter operand is set to zero. The addresses in general registers $R_1$ and $R_2$ are incremented by the amounts by which the corresponding length fields were reduced.

If the two operands, including the padding byte, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values.

At the completion of the operation, in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers $R_1$ and $R_2$ are set to zeros, even when one or both of the initial length values are zero. In any addressing mode, the contents of bit positions 0-39 of general registers $R_1 + 1$ and $R_2 + 1$ remain unchanged.

Access exceptions for the portion of a storage operand to the right of the first unequal byte may or may not be recognized. For operands longer than 2K bytes, access exceptions are not recognized more than 2K bytes beyond the byte being processed. Access exceptions are not indicated for locations more than 2K bytes beyond the first unequal byte.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the $R$ field associated with that operand is odd.

**Resulting Condition Code:**

- 0  Operands equal, or both zero length
- 1  First operand low
- 2  First operand high
- 3  --

**Program Exceptions:**

- Access (fetch, operands 1 and 2)
- Specification

**Programming Notes:**

1. An example of the use of the COMPARE LOGICAL LONG instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. When the $R_1$ and $R_2$ fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by the number of bytes compared, not by twice the number of bytes compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, condition code 0 is set. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed.

3. Special precautions should be taken when COMPARE LOGICAL LONG is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

4. Other programming notes concerning interruptible instructions are included in "Interruptible Instructions" in Chapter 5, "Program Execution."

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

**COMPARE LOGICAL LONG EXTENDED**

```
CLCLE R1,R3,D2(B2) [RS]
```

<table>
<thead>
<tr>
<th>'A9'</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The first operand is compared with the third operand until unequal bytes are compared, the
end of the longer operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with padding bytes. The result is indicated in the condition code.

The \( R_1 \) and \( R_3 \) fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and third operand is designated by the contents of general registers \( R_{\text{one}} \) and \( R_{\text{three}} \), respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers \( R_{\text{one}} + 1 \) and \( R_{\text{three}} + 1 \), respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the entire contents of general registers \( R_{\text{one}} + 1 \) and \( R_{\text{three}} + 1 \), respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers \( R_1 \) and \( R_3 \) is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers \( R_1 \) and \( R_3 \) constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The second-operand address is not used to address data; instead, the rightmost eight bits of the second-operand address, bits 56-63, are the padding byte. Bits 0-55 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-34 on page 7-62.
The comparison proceeds left to right, byte by byte, and ends as soon as an inequality is found, the end of the longer operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. If the operands are not of the same length, the shorter operand is
considered to be extended on the right with the appropriate number of padding bytes.

If both operands are of zero length, the operands are considered to be equal.

If the operation ends because of an inequality, the address fields in general registers \( R_1 \) and \( R_3 \) at completion identify the first unequal byte in each operand. The lengths in bit positions 32-63, in the 24-bit or 31-bit addressing mode, or in bit positions 0-63, in the 64-bit addressing mode, of general registers \( R_1 + 1 \) and \( R_3 + 1 \) are decremented by the number of bytes that were equal, unless the inequality occurred with the padding byte, in which case the length field for the shorter operand is set to zero. The addresses in general registers \( R_1 \) and \( R_3 \) are incremented by the amounts by which the corresponding length fields were decremented. Condition code 1 is set if the first operand is low, or condition code 2 is set if the first operand is high.

If the two operands, including the padding byte, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values. Condition code 0 is set.

If the operation is completed because a CPU-determined number of bytes have been compared without finding an inequality or reaching the end of the longer operand, the lengths in general registers \( R_1 + 1 \) and \( R_3 + 1 \) are decremented by the number of bytes compared, and the addresses in general registers \( R_1 \) and \( R_3 \) are incremented by the same number, so that the instruction, when reexecuted, resumes at the next bytes to be compared. If the operation is completed after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and the operand address is updated accordingly. Condition code 3 is set.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers \( R_1 \), \( R_1 + 1 \), \( R_3 \), and \( R_3 + 1 \), always remain unchanged.

The padding byte may be formed from \( D_2(B_2) \) multiple times during the execution of the instruction, and the registers designated by \( R_1 \) and \( R_3 \) may be updated multiple times. Therefore, if \( B_2 \) equals \( R_1 \), \( R_1 + 1 \), \( R_3 \), or \( R_3 + 1 \) and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed. The maximum amount is approximately 4K bytes of either operand.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers \( R_1 \) and \( R_3 \) may be set to zeros or may remain unchanged from their original values, even when one or both of the initial length values are zero.

Access exceptions for the portion of a storage operand to the right of the first unequal byte may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the byte being processed. Access exceptions are not indicated for locations more than 4K bytes beyond the first unequal byte.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

**Resulting Condition Code:**

- 0 All bytes compared; operands equal, or both zero length
- 1 All bytes compared, first operand low
- 2 All bytes compared, first operand high
- 3 CPU-determined number of bytes compared without finding an inequality

**Program Exceptions:**

- Access (fetch, operands 1 and 3)
- Specification

**Programming Notes:**

1. COMPARE LOGICAL LONG EXTENDED is intended for use in place of COMPARE LOGICAL LONG when the operand lengths are specified as 32-bit binary integers. COMPARE LOGICAL LONG EXTENDED sets condition code 3 in cases in which COMPARE LOGICAL LONG would be interrupted.
2. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of bytes that were compared.

3. The function of not processing more than approximately 4K bytes of either operand is intended to permit software polling of a flag that may be set by a program on another CPU during long operations.

4. When the R<sub>1</sub> and R<sub>3</sub> fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by the number of bytes compared, not by twice the number of bytes compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, the condition code is finally set to 0 after possible settings to 3. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed. If storage is not accessed, condition code 3 may or may not be set regardless of the operand length.

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

**COMPARE LOGICAL LONG UNICODE**

<table>
<thead>
<tr>
<th>CLCLU</th>
<th>R&lt;sub&gt;1&lt;/sub&gt;, R&lt;sub&gt;3&lt;/sub&gt;, D&lt;sub&gt;2&lt;/sub&gt;(B&lt;sub&gt;2&lt;/sub&gt;)</th>
<th>[RSY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EB'</td>
<td>R&lt;sub&gt;1&lt;/sub&gt; R&lt;sub&gt;3&lt;/sub&gt; B&lt;sub&gt;2&lt;/sub&gt; DL&lt;sub&gt;2&lt;/sub&gt; DH&lt;sub&gt;2&lt;/sub&gt; '8F'</td>
<td></td>
</tr>
<tr>
<td>0 8 12 16 20 32 40 47</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first operand is compared with the third operand until unequal two-byte Unicode characters are compared, the end of the longer operand is reached, or a CPU-determined number of characters have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with two-byte padding characters. The result is indicated in the condition code.

The R<sub>1</sub> and R<sub>3</sub> fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost character of the first operand and third operand is designated by the contents of general registers R<sub>1</sub> and R<sub>3</sub>, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers R<sub>1</sub> + 1 and R<sub>3</sub> + 1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 0-63 of general registers R<sub>1</sub> + 1 and R<sub>3</sub> + 1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The contents of general registers R<sub>1</sub> + 1 and R<sub>3</sub> + 1 must specify an even number of bytes; otherwise, a specification exception is recognized.

The handling of the addresses in general registers R<sub>1</sub> and R<sub>3</sub> is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R<sub>1</sub> and R<sub>3</sub> constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of the registers constitute the address.

The second-operand address is not used to address data; instead, the rightmost 16 bits of the second-operand address, bits 48-63, are the two-byte padding character. Bits 0-47 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-35 on page 7-65.

The comparison proceeds left to right, character by character, and ends as soon as an inequality is found, the end of the longer operand is reached, or a CPU-determined number of characters have been compared, whichever occurs first. If the operands are not of the same length, the shorter operand is considered to be extended on the right.
<table>
<thead>
<tr>
<th>Register</th>
<th>24-Bit Addressing Mode</th>
<th>31-Bit Addressing Mode</th>
<th>64-Bit Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>/---------/</td>
<td>First-Operand Address</td>
<td>/--------/</td>
</tr>
<tr>
<td></td>
<td>0 0 40</td>
<td></td>
<td>0 0 33</td>
</tr>
<tr>
<td>$R_1 + 1$</td>
<td>/--------/</td>
<td>First-Operand Length</td>
<td>/--------/</td>
</tr>
<tr>
<td></td>
<td>0 0 32</td>
<td></td>
<td>0 0 32</td>
</tr>
<tr>
<td>$R_3$</td>
<td>/---------/</td>
<td>Third-Operand Address</td>
<td>/--------/</td>
</tr>
<tr>
<td></td>
<td>0 0 40</td>
<td></td>
<td>0 0 33</td>
</tr>
<tr>
<td>$R_3 + 1$</td>
<td>/--------/</td>
<td>Third-Operand Length</td>
<td>/--------/</td>
</tr>
<tr>
<td></td>
<td>0 0 32</td>
<td></td>
<td>0 0 32</td>
</tr>
<tr>
<td>2nd Op. Address</td>
<td>/---------/</td>
<td>Pad</td>
<td>/--------/</td>
</tr>
<tr>
<td></td>
<td>0 0 48</td>
<td></td>
<td>0 0 48</td>
</tr>
</tbody>
</table>

**Figure 7-35. Register Contents and Second-Operand Address for COMPARE LOGICAL LONG UNICODE**

with the appropriate number of two-byte padding characters.

If both operands are of zero length, the operands are considered to be equal.
If the operation ends because of an inequality, the address fields in general registers \( R_1 \) and \( R_3 \) at completion identify the first unequal two-byte character in each operand. The lengths in bit positions 32-63, in the 24-bit or 31-bit addressing mode, or in bit positions 0-63, in the 64-bit addressing mode, of general registers \( R_1 + 1 \) and \( R_3 + 1 \) are decremented by 2 times the number of characters that were equal, unless the inequality occurred with the two-byte padding character, in which case the length field for the shorter operand is set to zero. The addresses in general registers \( R_1 \) and \( R_3 \) are incremented by the amounts by which the corresponding length fields were decremented. Condition code 1 is set if the first operand is low, or condition code 2 is set if the first operand is high.

If the two operands, including the two-byte padding character, if necessary, are equal, both length fields are made zero at completion, and the addresses are incremented by the corresponding operand-length values. Condition code 0 is set.

If the operation is completed because a CPU-determined number of characters have been compared without finding an inequality or reaching the end of the longer operand, the lengths in general registers \( R_1 + 1 \) and \( R_3 + 1 \) are decremented by 2 times the number of characters compared, and the addresses in general registers \( R_1 \) and \( R_3 \) are incremented by the same number, so that the instruction, when reexecuted, resumes at the next characters to be compared. If the operation is completed after the shorter operand has been exhausted, the length field pertaining to the shorter operand is zero, and the operand address is updated accordingly. Condition code 3 is set.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers \( R_1 \), \( R_1 + 1 \), \( R_2 \), and \( R_2 + 1 \), always remain unchanged.

The two-byte padding character may be formed from \( D_2(B_2) \) multiple times during the execution of the instruction, and the registers designated by \( R_1 \) and \( R_3 \) may be updated multiple times. Therefore, if \( B_2 \) equals \( R_1 \), \( R_1 + 1 \), \( R_3 \), or \( R_3 + 1 \) and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers \( R_1 \) and \( R_3 \) may be set to zeros or may remain unchanged from their original values, including the case when one or both of the initial length values are zero.

Access exceptions for the portion of a storage operand to the right of the first unequal character may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized more than 4K bytes beyond the character being processed. Access exceptions are not indicated for locations more than 4K bytes beyond the first unequal character.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field or length associated with that operand is odd.

**Resulting Condition Code:**

0  All characters compared; operands equal, or both zero length  
1  First operand low  
2  First operand high  
3  CPU-determined number of characters compared without finding an inequality

**Program Exceptions:**

- Access (fetch, operands 1 and 3)  
- Operation (if the extended-translation facility 2 is not installed)  
- Specification

**Programming Notes:**

1. COMPARE LOGICAL LONG UNICODE is intended for use in place of COMPARE LOGICAL LONG or COMPARE LOGICAL LONG EXTENDED when two-byte characters are to be compared. The characters may be Unicode characters or any other double-byte characters. COMPARE LOGICAL LONG UNICODE sets condition code 3 in cases in which COMPARE LOGICAL LONG would be interrupted.
2. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of characters that were compared.

3. When the $R_1$ and $R_3$ fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, except that the contents of the designated registers are incremented or decremented only by 2 times the number of characters compared, not by 4 times the number of characters compared. In the absence of dynamic modification of the operand area by another CPU or by a channel program, the condition code is finally set to 0 after possible settings to 3. However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed. If storage is not accessed, condition code 3 may or may not be set regardless of the operand length.

4. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

5. If padding with a Unicode space character is required (or any character whose representation is less than or equal to FFF hex), the character may be represented in the displacement field of the instruction, for example:

```
CLCLU 6,8,X'020'
```

### COMPARE LOGICAL STRING

```
CLST R1,R2 [RRE]

<table>
<thead>
<tr>
<th>'B25D'</th>
<th>//////////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>
```

The first operand is compared with the second operand until unequal bytes are compared, the end of either operand is reached, or a CPU-determined number of bytes have been compared, whichever occurs first. The CPU-determined number is at least 256. The result is indicated in the condition code.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers $R_1$ and $R_2$, respectively.

The handling of the addresses in general registers $R_1$ and $R_2$ is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers $R_1$ and $R_2$ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The first and second operands may be of the same or different lengths. The end of an operand is indicated by an ending character in the last byte position of the operand. The ending character to be used to determine the end of an operand is specified in bit positions 56-63 of general register 0. Bit positions 32-55 of general register 0 are reserved for possible future extensions and must contain all zeros; otherwise, a specification exception is recognized.

The operation proceeds left to right, byte by byte, and ends as soon as the ending character is encountered in either or both operands, unequal bytes which do not include an ending character are compared, or a CPU-determined number of bytes have been compared, whichever occurs first. The CPU-determined number is at least 256. When the ending character is encountered simultaneously in both operands, including when it is in the first byte position of the operands, the operands are of the same length and are considered to be equal, and condition code 0 is set. When the ending character is encountered in only one operand, that operand, which is the shorter operand, is considered to be low, and condition code 1 or 2 is set. Condition code 1 is set if the first operand is low, or condition code 2 is set if the second operand is low. Similarly, when unequal bytes which do not include an ending character are compared, condition code 1 is set if the lower byte is in the first operand, or condition code 2 is set if the lower byte is in the second operand. When a CPU-determined number of bytes have been compared, condition code 3 is set.

When condition code 1 or 2 is set, the address of the last byte processed in the first and second
operands is placed in general registers $R_1$ and $R_2$, respectively. That is, when condition code 1 is set, the address of the ending character or first unequal byte in the first operand, whichever was encountered, is placed in general register $R_1$, and the address of the second-operand byte corresponding in position to the first-operand byte is placed in general register $R_2$. When condition code 2 is set, the address of the ending character or first unequal byte in the second operand, whichever was encountered, is placed in general register $R_2$, and the address of the first-operand byte corresponding in position to the second-operand byte is placed in general register $R_1$. When condition code 3 is set, the address of the next byte to be processed in the first and second operands is placed in general registers $R_1$ and $R_2$, respectively. Whenever an address is placed in a general register, bits 32-39 of the register, in the 24-bit addressing mode, or bit 32, in the 31-bit addressing mode, are set to zeros. Bits 0-31 of the $R_1$ and $R_2$ registers always remain unchanged in the 24-bit or 31-bit mode.

When condition code 0 is set, the contents of general registers $R_1$ and $R_2$ remain unchanged.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

Access exceptions for the first and second operands are recognized only for that portion of the operand which is necessarily examined in the operation.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Entire operands equal; general registers $R_1$ and $R_2$ unchanged</td>
</tr>
<tr>
<td>1</td>
<td>First operand low; general registers $R_1$ and $R_2$ updated with addresses of last bytes processed</td>
</tr>
<tr>
<td>2</td>
<td>First operand high; general registers $R_1$ and $R_2$ updated with addresses of last bytes processed</td>
</tr>
<tr>
<td>3</td>
<td>CPU-determined number of bytes equal; general registers $R_1$ and $R_2$ updated with addresses of next bytes</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Access (fetch, operands 1 and 2)
- Specification

**Programming Notes:**

1. Several examples of the use of the COMPARE LOGICAL STRING instruction are given in Appendix A, "Number Representation and Instruction-Use Examples."

2. When condition code 0 is set, no indication is given of the position of either ending character.

3. When condition code 3 is set, the program can simply branch back to the instruction to continue the comparison. The program need not determine the number of bytes that were compared.

4. $R_1$ or $R_2$ may be zero, in which case general register 0 is treated as containing an address and also the ending character.

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

**COMPARE UNTIL SUBSTRING EQUAL**

CUSE $R_1,R_2$ [RRE]

<table>
<thead>
<tr>
<th>'B257'</th>
<th>/////</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

The first operand is compared with the second operand until equal substrings (sequences of bytes) of a specified length are found, the end of the longer operand is reached, or a CPU-determined number of unequal bytes have been compared, whichever occurs first. The shorter operand is considered to be extended on the right with padding bytes. The CPU-determined number is at least 256. The result is indicated in the condition code.

The $R_1$ and $R_2$ fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is specified by the
contents of the \( R_1 \) and \( R_2 \) general registers, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the 32-bit signed binary integer in bit positions 32-63 of general registers \( R_1 + 1 \) and \( R_2 + 1 \), respectively. In the 64-bit addressing mode, the number of bytes is specified by the 64-bit signed binary integer in bit positions 0-63 of those registers. When an operand length is negative, it is treated as zero, and it remains unchanged upon completion of the instruction.

Bits 56-63 of general register 0 specify the unsigned substring length, a value of 0-255, in bytes. Bits 56-63 of general register 1 are the padding byte. Bits 0-55 of general registers 0 and 1 are ignored.

The handling of the addresses in general registers \( R_1 \) and \( R_2 \) is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers \( R_1 \) and \( R_2 \) constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-36 on page 7-70.
The result is obtained as if the operands were processed from left to right. However, multiple accesses may be made to all or some of the bytes of each operand.

The comparison proceeds left to right, byte by byte, and ends as soon as (1) equal substrings of the specified length are found, (2) the end of the longer operand is reached without finding equal substrings of the specified length, or (3) the last bytes compared are unequal, and a
CPU-determined number of bytes have been compared. The CPU-determined number is at least 256. If the operands are not of the same length, the shorter operand is considered to be extended on the right with the appropriate number of padding bytes.

If the operation ends because equal substrings of the specified length were found, the condition code is set to 0. If the operation ends because the end of the longer operand was reached without finding equal substrings of the specified length, the condition code is set to 1 if equal bytes were the last bytes compared, or it is set to 2 if unequal bytes were the last bytes compared. If the operation ends because unequal bytes were compared when a CPU-determined number of bytes had been compared, the condition code is set to 3.

If the specified substring length is zero, it is considered that equal substrings of the specified length were found, and condition code 0 is set.

If both operands are of zero length but the specified substring length is not zero, it is considered that the end of the longer operand was reached when unequal bytes were the last bytes compared, and condition code 2 is set.

If equal bytes have been compared but then unequal bytes are compared, it is considered that all bytes so far compared are unequal.

At the completion of the operation, the operand-length fields in the \( R_1 + 1 \) and \( R_2 + 1 \) registers are decremented by the number of unequal bytes compared (including equal bytes before unequal bytes compared), and the addresses in the \( R_1 \) and \( R_2 \) registers are incremented by the same number. However, in the case when a byte of the longer operand is compared against the padding byte, the length field for the shorter operand is not decremented below zero, and the corresponding address is not incremented above the address of the first byte after the shorter operand. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the addresses in bit positions 32-63 of registers \( R_1 \) and \( R_2 \) are set to zeros, even if the substring length is zero or both operand lengths are initially zero.

Thus, when condition code 0 or 1 is set, the resulting addresses in the \( R_1 \) and \( R_2 \) registers designate the first bytes of equal substrings in the two operands, and the lengths in the \( R_1 + 1 \) and \( R_2 + 1 \) registers have been decremented by the number of bytes preceding the equal substrings, except when the equal substring in the shorter operand begins with the padding byte, in which case the length field for the shorter operand is zero, and the corresponding address field has been incremented by the operand length. When condition code 2 is set, each address field designates the first byte after the corresponding operand, and both length fields are zero. When condition code 3 is set, each address field designates the first byte after the last compared byte of the corresponding operand, and both length fields have been decremented by the number of bytes compared, except that a length field is not decremented below zero.

When the contents of the \( R_1 \) and \( R_2 \) fields are the same, the first and second operands may be compared, or the condition code may be set to 0 or 1 without comparing the operands.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers \( R_1 \), \( R_1 + 1 \), \( R_2 \), and \( R_2 + 1 \), always remain unchanged.

The substring length and padding byte may be fetched from general registers 0 and 1 multiple times during the execution of the instruction, and the registers designated by \( R_1 \) and \( R_2 \) may be updated multiple times. Therefore, if \( R_1 \) or \( R_2 \) is zero, the results are unpredictable.

When condition code 3 is set, the general registers used by the instruction have been set so that the remainder of the operands can be processed by simply branching back and reexecuting the instruction.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

The execution of the instruction is interruptible when the last bytes compared are unequal; it is not interruptible when the last bytes compared are equal. When an interruption occurs, other than one that follows termination, the contents of the registers designated by the \( R_1 \) and \( R_2 \) fields are
updated the same as upon normal completion of the instruction, so that the instruction, when reexecuted, resumes at the point of interruption. The condition code is unpredictable.

Access exceptions for the portion of a storage operand to the right of the last byte processed may or may not be recognized. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Although the operand address and length fields remain unchanged when a zero substring length is specified, the recognition of access exceptions is not necessarily prevented.

**Resulting Condition Code:**

0  Equal substrings of specified length found
1  End of longer operand reached when last bytes compared are equal
2  End of longer operand reached when last bytes compared are unequal
3  Last bytes compared are unequal, and CPU-determined number of bytes compared

**Program Exceptions:**

- Access (fetch, operands 1 and 2)
- Specification

**Programming Notes:**

1. When the R₁ and R₂ fields are the same, the operation proceeds in the same way as when two distinct pairs of registers having the same contents are specified, and, in the absence of dynamic modification of the operand area by another CPU or by a channel program, condition code 0, 1, or 2 is set (as explained in the next note). However, it is unpredictable whether access exceptions are recognized for the operand since the operation can be completed without storage being accessed.

2. If the contents of the R₁ and R₂ fields are the same and the operand length is nonzero, and provided that another CPU or a channel program is not changing an operand, condition code 0 is set if the operand length is equal to or greater than the specified substring length, or condition code 1 is set if the operand length is less than the specified substring length. Whether or not R₁ equals R₂, if both operand lengths are zero, condition code 0 is set if the specified substring length is zero, or condition code 2 is set if the specified substring length is nonzero. In all of these cases, the addresses in the R₁ and R₂ registers and the lengths in the R₁ + 1 and R₂ + 1 registers remain unchanged.

3. Special precautions should be taken when COMPARE UNTIL SUBSTRING EQUAL is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

4. Other programming notes concerning interruptible instructions are included in "Interruptible Instructions" on page 5-21.

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

6. The storage-operand references of COMPARE UNTIL SUBSTRING EQUAL may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98.)

**COMPRESSION CALL**

<table>
<thead>
<tr>
<th>CMPSC</th>
<th>R₁, R₂</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B263'</td>
<td>//////////</td>
<td>R₁  R₂</td>
</tr>
<tr>
<td>0 16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This definition assumes knowledge of the introductory information and information about dictionary formats in *Enterprise Systems Architecture/390 Data Compression*, SA22-7208-01.

The second operand is compressed or expanded, depending on a specification in general register 0, and the results are placed at the first-operand location. The compressed-data operand normally consists of index symbols corresponding to entries in a dictionary designated by an address in general register 1. This dictionary is a compression dictionary during a compression operation or an expansion dictionary during an expansion operation. During compression when format-1 sibling descriptors are specified in general register 0, an expansion dictionary immediately follows the compression dictionary. During
compression when the symbol-translation option is specified in general register 0, the index symbols resulting from compression are translated to interchange symbols by means of a symbol-translation table designated by the address and an offset in general register 1, and it is the interchange symbols that are placed at the first-operand location. The number of bits in a symbol in the compressed-data operand is specified in general register 0. The operation proceeds until the end of either operand is reached or a CPU-determined amount of data has been processed, whichever occurs first. The results are indicated in the condition code.

The R₁ and R₂ fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte containing any bit of the first operand and second operand is designated by an address in general registers R₁ and R₂, respectively. The number of bytes containing any bits of the first operand and second operand is specified by bits 32-63 of general registers R₁ + 1 and R₂ + 1, respectively, in the 24-bit or 31-bit addressing mode or by bits 0-63 of the registers in the 64-bit addressing mode. The contents of general registers R₁ + 1 and R₂ + 1 are treated as 32-bit unsigned binary integers in the 24-bit or 31-bit addressing mode or as 64-bit unsigned binary integers in the 64-bit addressing mode.

The location of the leftmost byte of the compression dictionary during compression, or of the expansion dictionary during expansion, is designated on a 4K-byte boundary by an address in general register 1.

The handling of the addresses in general registers R₁, R₂, and 1 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of registers R₁ and R₂ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of registers R₁ and R₂ constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of registers R₁ and R₂ constitute the address. In the 24-bit addressing mode, the contents of bit positions 40-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 0-39 and 52-63 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 0-32 and 52-63 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-51 of register 1, with 12 rightmost zeros appended, constitute the address, and the contents of bit positions 52-63 are ignored.

Although the contents of bit positions 52-63 of general register 1 are ignored as just described, those contents are used as follows. The contents of bit positions 61-63 of the register are the compressed-data bit number (CBN). At the beginning of the operation, the CBN designates the leftmost bit within the leftmost byte of the compressed-data operand. The compressed-data operand is the first operand during compression, or it is the second operand during expansion. When the symbol-translation option is specified during compression, the contents of bit positions 52-60 of the register, with seven rightmost zeros appended, are the byte offset from the beginning of the compression dictionary to the leftmost byte of the symbol-translation table. Symbol translation cannot be specified during expansion, and the contents of bit positions 52-60 are ignored during expansion.

The contents of the registers just described and also of general register 0 are shown in Figure 7-37 on page 7-74.

Bit 55 (E) of general register 0 specifies the compression operation if zero or the expansion operation if one.

Bit 47 (ST) of general register 0 is the symbol-translation-option bit. During compression when bit 47 is zero, the operation produces indexes, called index symbols, to compression-dictionary entries that represent character strings, and the operation then places the index symbols in the compressed-data operand. During compression when bit 47 is one, the operation still produces index symbols but then translates the index symbols to other symbols, called interchange symbols, that it then places in the compressed-data operand. This symbol translation is done by using the symbol-translation table specified by the address and offset in general register 1. Bit 47
and the offset in general register 1 are ignored during expansion. During expansion, the compressed-data operand always contains index symbols that designate entries in the expansion dictionary.

Bits 48-51 (CDSS) of general register 0 specify the number of bits in the index symbols or interchange symbols in the compressed-data operand, as shown in the figure. Bits 48-51 must not have any of the values 0000 or 0110-1111 binary; otherwise, a specification exception is recognized. When symbol-translation is not specified, bits
64-Bit Addressing Mode (Continued)

When ST Bit Is Zero during Compression, or during Expansion

\[
\begin{array}{c|c|c|c|c|c}
\text{GR1} & \text{Dictionary Origin}^1 & \text{STT Off.} & \text{CBN} \\
\hline
0 & 52 & 61 & 63 \\
\end{array}
\]

When ST Bit Is One during Compression

\[
\begin{array}{c|c|c|c|c|c}
\text{GR1} & \text{Dictionary Origin}^1 & \text{STT Off.} & \text{CBN} \\
\hline
0 & 52 & 61 & 63 \\
\end{array}
\]

Explanation:

1. Compression dictionary during compression, or expansion dictionary during expansion
2. CBN Compressed-data bit number
3. CDSS Compressed-data symbol size, and dictionary size when symbol translation not specified
   - CDSS Symbol Dictionary
     - (binary) Size Size
     - 0000 Causes a specification exception to be recognized
     - 0001 9 bits 512 entries, 4K bytes
     - 0010 10 bits 1K entries, 8K bytes
     - 0011 11 bits 2K entries, 16K bytes
     - 0100 12 bits 4K entries, 32K bytes
     - 0101 13 bits 8K entries, 64K bytes
     - 0110-1111 Causes a specification exception to be recognized
4. E Expansion operation
5. F1 Format-1 sibling descriptors (ignored during expansion)
6. ST Symbol-translation option (ignored during expansion)
7. STT Off. Symbol-translation-table offset (ignored during expansion)

Figure 7-37 (Part 2 of 2). Register Contents for COMPRESSION CALL

48-51 also specify, as shown in the figure, the number of eight-byte entries in each of the compression and expansion dictionaries, and, thus, they specify the size in bytes of each of the dictionaries. When symbol translation is specified, the compression dictionary is considered to extend to the beginning of the symbol-translation table, that is, the size in bytes of the compression dictionary is the offset in bit positions 52-60 of general register 1, with seven rightmost zeros appended. The size in bytes of the symbol-translation table is considered to be one fourth that of the compression dictionary. However, the offset in general register 1 must be at least as large as the size of the compression dictionary would be if symbol translation were not specified and the CDSS were one less than it actually is, and, when the CDSS is 0001 binary, the offset must be at least 2K bytes; otherwise, the results are unpredictable. For example, if the CDSS is 0101, the offset must be at least 32K bytes.

Bit 54 (F1) of general register 0 specifies that the compression dictionary contains format-0 sibling descriptors if the bit is zero or format-1 sibling descriptors if the bit is one. Sibling descriptors are used during the compression operation. A format-0 sibling descriptor is eight bytes at an index position in the compression dictionary. A format-1 sibling descriptor is 16 bytes, with the first eight bytes at an index position in the compression dictionary and the second eight bytes at
the same index position in the expansion dictionary. During compression when bit 54 is one, an expansion dictionary is considered to immediately follow the compression dictionary specified by the address in general register 1. Bit 54 is ignored during expansion.

Bits 47 and 54 of general register 0 must not both be ones; otherwise, the results are unpredictable.

The unused bit positions in general register 0 are reserved for possible future extensions and should contain zeros; otherwise, the program may not operate compatibly in the future.

In the access-register mode, the contents of access register $R_1$ are used for accessing the first operand, and the contents of access register $R_2$ are used for accessing the second operand and the dictionaries and the symbol-translation table.

The operation starts at the left end of both operands and proceeds to the right. The operation is ended when the end of either operand is reached or when a CPU-determined amount of data has been processed, whichever occurs first.

During a compression operation, the end of the first operand is considered to be reached when the number of unused bit positions remaining in the first-operand location is not sufficient to contain additional compressed data.

During an expansion operation, the end of the first-operand location is considered to be reached when either of the following two conditions is met:

1. The number of unused byte positions remaining in the first-operand location is not sufficient to contain all the characters that would result from expansion of the next index symbol.
2. Immediately when the number of unused byte positions is zero, that is, immediately when the expansion of an index symbol completely fills the first-operand location.

During an expansion operation, the end of the second-operand location is considered to be reached when the next index symbol does not reside entirely within the second-operand location. The second-operand location ends at the beginning of the byte designated by the sum of the address in general register $R_2$ and the length in general register $R_2 + 1$, regardless of the compressed-data bit number in bit positions 61-63 of general register 1.

If the operation is ended because the end of the second operand is reached, condition code 0 is set. If the operation is ended because the end of the first operand is reached, condition code 1 is set, except that condition code 0 is set if the end of the second operand is also reached. If the operation is ended because a CPU-determined amount of data has been processed, condition code 3 is set.

At the completion of the operation, the length in general register $R_1 + 1$ is decremented by the number of complete bytes stored at the first-operand location, and the address in general register $R_1$ is incremented by the same amount. During compression, a complete byte is considered to be stored only if all of its bit positions contain bits of compressed data. During compression when the first bit of compressed data stored is not in bit position 0 of a byte, the bits in the byte to the left of the first bit of compressed data remain unchanged. During compression if the last byte stored does not completely contain compressed data, the bits in the byte to the right of the rightmost bit of compressed data in the byte either are unchanged or are set to zeros.

The length in general register $R_2 + 1$ is decremented by the number of complete bytes processed at the second-operand location, and the address in general register $R_2$ is incremented by the same amount. During expansion, a complete byte is considered to be processed only if all of its bits have been used to produce expanded data.

The leftmost bits which are not part of the address in general registers $R_1$ and $R_2$ may be set to zeros or may remain unchanged. However, in the 24-bit or 31-bit addressing mode, bits 0-31 of these registers and also of general registers $R_1 + 1$ and $R_2 + 1$ always remain unchanged.

The bit number of the bit following the last bit of compressed data processed is placed in bit positions 61-63 of general register 1, and bits 52-60 of the register and the leftmost bits which are not part of the address in the register may be set to zeros or may remain unchanged, except that when one or both of the original length values are so small that no compressed data can be proc-
essed, all bits in the register may remain unchanged. However, when bit 47 of general register 0 is one, bits 52-60 of general register 1 always remain unchanged. Also, in the 24-bit or 31-bit addressing mode, bits 0-31 of the register always remain unchanged.

If the operands overlap one another or the first operand overlaps the dictionaries or the symbol-translation table in storage in any way, the results are unpredictable.

When symbol translation is specified, the symbol-translation table consists of two-byte entries, and an entry contains an interchange symbol in the rightmost bit positions of the entry. The length of the interchange symbol is specified by bits 48-51 of general register 0. The left-hand bits that are not part of the interchange symbol in a symbol-translation-table entry must be zeros; otherwise, the results are unpredictable.

To translate an index symbol to an interchange symbol, the index symbol is multiplied by 2 and then added to the address of the beginning of the symbol-translation table to locate an entry in the table, and then the interchange symbol is obtained from the entry.

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the contents of the registers designated by the R₁ and R₂ fields and of general register 1 are updated the same as upon normal completion of the instruction, so that the instruction, when reexecuted, resumes at the point of interruption. The condition code is unpredictable.

For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions may be recognized for all locations in the dictionaries and symbol-translation table if those areas are specified to be used and even if the locations would not be used during the operation. Access exceptions are not recognized for an operand, the dictionaries, or the symbol-translation table if the R field associated with that operand is odd. Also, when the R₁ field is odd, PER storage-alteration events are not recognized, and no change bits are set.

If an access exception is due to be recognized for either of the operands or for a dictionary or the symbol-translation table, the result is that either the exception is recognized or condition code 3 is set. If condition code 3 is set, the exception will be recognized when the instruction is executed again to continue processing the same operands, assuming that the exception condition still exists.

During compression, regardless of whether the exception is recognized or condition code 3 is set, a nullifying access-exception condition or a suppressing page-protection exception condition is handled so that an index symbol is generated only if it is the one that would result if there were no access-exception condition.

During compression or expansion, regardless of whether the exception is recognized or condition code 3 is set, a nullifying or suppressing access-exception condition may result in data having been stored at the first-operand location at or to the right of the location designated by the final address in general register R₁, which result is not true nullification or suppression. The amount of data stored depends on the reason for the access-exception condition. If the condition is due to a reference to a dictionary or the symbol-translation table, up to 4K bytes of data may have been stored at or to the right of the location designated by the final address. If the condition is due to a reference to the first or second operand, part of one index or interchange symbol, during compression, or part of one character symbol, during expansion, may have been stored at or to the right of the location designated by the final address. In all cases, the storing will be repeated when the instruction is executed again to continue processing the same operands.

If the end of the first operand is reached and an access exception is due to be recognized for the second operand, it is unpredictable whether condition code 1 is set or the access exception is recognized.

During expansion when the expansion dictionary is not logically correct, unusual storing may occur as described in the section “Expansion Process” in Chapter 1 of Enterprise Systems Architecture/390 Data Compression, SA22-7208-01. The results of an access exception in this case may not be true nullification or suppression.

Special Conditions
During compression of each character symbol, either the characters in the symbol or the dictionary character entries (not sibling descriptors) representing characters of the symbol are counted, and a data exception is recognized if this count becomes too large. The count can reach at least 260 without the exception being recognized.

During compression, the number of child characters or sibling characters processed during the processing of each parent entry are counted, and a data exception is recognized if this count becomes too large. The count can reach at least 260 without the exception being recognized. That is, a parent must not have more than 260 children; otherwise, a data exception may be recognized.

During expansion of each character symbol, either the characters in the symbol or the dictionary entries representing characters of the symbol are counted, and a data exception is recognized if this count becomes too large. If the characters in the symbol are counted, the count can reach at least 260 without the exception being recognized. If the dictionary entries representing characters of the symbol are counted, the count can reach at least 127 without the exception being recognized.

Certain error conditions in the dictionaries cause a data exception to be recognized and the operation to be terminated. Some of these error conditions are described in the sections “Expansion Process” and “Results of Dictionary Errors” in Chapter 1 of SA22-7208-01. The others are described in Chapter 2 of SA22-7208-01.

**Resulting Condition Code:**

0  End of second operand reached
1  End of first operand reached and end of second operand not reached
2  --
3  CPU-determined amount of data processed

**Program Exceptions:**

- Access (fetch, operand 2, dictionaries, and symbol-translation table; store, operand 1)
- Data
- Specification

**Programming Notes:**

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the operation. The program need not determine the amount of data processed.

2. During compression when a nullifying access exception is due to be recognized, an index symbol is generated only if it is the one that would result if there were no access-exception condition. The result of this is that compression of the same expanded data by means of one or more executions of the instruction and by using the same dictionary always results in the same compressed data. That is, (1) the best possible matches in the dictionary are always found for the characters in the second operand, or else the execution is ended by either setting CC3 or recognizing the exception, and (2) the results of compression are repeatable (although possibly by means of a different number of executions of the instruction) and predictable.

For example, if the next characters of the string being compressed are ABC, and dictionary entry A has a child B, which has a child C, the normal operation is to compress ABC as a single index symbol, but if the C of the string is in the first byte of an invalid page (a page-translation exception is due to be recognized), no index symbol is generated. More specifically, an index symbol corresponding to the character symbol AB is not generated because this is not the index symbol that would be generated if the access-exception condition did not exist.

In the above, “best possible match” refers only to the characters in the second operand. In the example above, if the next two characters of the second operand (the string) are AB, and these are the last two characters of the second operand, the best possible match is on AB, even though there could be a match on ABC if the second operand included one more byte containing C.

Expansion is normally always repeatable. An index symbol is always expanded to exactly the character symbol it represents unless an exception that causes termination is recognized.

3. During expansion, if at least one unused byte position remains in the first operand location,
COMPRESSION CALL may completely process the next index symbol in the second operand before it determines that the first-operand location does not have sufficient unused byte positions to contain the expanded data that would result from the next index symbol. If that next index symbol causes encountering of bad dictionary entries, the result can be either a data exception or condition code 1.

COMPRESSION CALL immediately sets condition code 1 when processing of an index symbol exactly fills the first-operand location, except that it sets condition code 0 if the end of the second-operand location also has been reached. Immediately setting condition code 1 has the advantage that data can be compressed using one dictionary and then followed immediately, possibly on a bit boundary, by a different type of data compressed using another dictionary. The compressed data can be successfully expanded if, during the expansion of the data compressed using the first dictionary, the length of the first-operand location is specified to be exactly the length of the expanded data that will be produced. Condition code 1 will then be set when the first-operand location is full, at which time the specification of the dictionary can be changed in order to expand the remainder of the compressed data using the second dictionary. If the definition allowed condition code 1 not to be set, it might be attempted to expand the next index symbol, which resulted from use of the second dictionary, by means of the first dictionary, and this might cause recognition of a data exception. For example, the next index symbol, which properly designates a character entry in the second dictionary, might designate the second half of a format-1 sibling descriptor in the first dictionary, and that second half might begin with a character, such as 0 (F0 hex), that would appear to be an invalid partial symbol length in a character entry.

4. A nullifying access-exception condition due to a reference to a dictionary or the symbol-translation table may result in the storing of data at or to the right of the location designated by the final address in general register R1. This storing and the processing needed to produce the data stored will be repeated when COMPRESSION CALL is executed again to continue processing the same operands. The repeated processing will reduce the performance of the instruction execution, and it should be avoided by ensuring that the environment in which the program is executed is one in which page-translation-exception conditions for the dictionaries and symbol-translation table are infrequent.

5. Following is an example of how the compressed-data bit number (CBN) is used and set. In this example:

- The operation is an expansion operation.
- The CDSS in general register 0 is 0010 binary. Therefore, there are 1K entries in the expansion dictionary, and the length of an index symbol is 10 bits.
- The second operand (compressed-data operand) begins at location 6000 hex and has a length of five bytes. The initial CBN is 7. Therefore, there are three index symbols to be expanded, and the final CBN will be 5.
- The compressed data beginning at location 6000 hex is 0081FF9FF8 hex. Therefore, the three index symbols are 103, 3FC, and 3FF hex.
- The first operand (expanded-data operand) begins at location 5000 hex and has a length of 64 bytes. The three index symbols are expanded to a total of 14 bytes of expanded data.

The following figure shows the initial and final contents of general registers R1, R1 + 1, R2, and R2 + 1, the contents of locations 6000-6004 hex in binary, and the way a cursor corresponding to the CBN is advanced during the expansion operation.
6. The reason for allowing a parent to have no more than 260 children is as follows. The parent can contain five identical child characters. Then, 255 different sibling characters are possible — all of these must be different from the child characters and each other, or else they may be wasted (never matched against), depending on the implementation. Thus, every possible child is permitted.

7. Symbol translation is for use by VTAM. VTAM will begin by doing compression by means of software and an adaptive dictionary. When the adaptive dictionary has matured such that the degree of compression becomes sufficiently good (crosses some threshold), VTAM will “freeze” (stop adapting) its dictionary, inform the other end of the session to freeze also, transform its adaptive dictionary to the dictionary form used by COMPRESSION CALL, and then use COMPRESSION CALL to continue on with the compression. The other end of the session can continue to use its frozen adaptive dictionary.

Following is clarification about the STT offset. Assume VTAM uses a 4K-entry adaptive dictionary, which is the largest size VTAM uses. All of the entries in this dictionary correspond to character symbols because there are no sibling descriptors in the VTAM dictionary. The VTAM dictionary cannot map one-to-one to a COMPRESSION CALL dictionary because the latter requires that some of the entries be sibling descriptors. Therefore, VTAM must have an 8K-entry dictionary for use in the basic compression operation. Only the first hundred or so entries in the second 4K of the 8K need to be used, and these entries compensate for (take the place of) the entries in the first 4K that must be sibling descriptors. The STT can and should, to save space, begin immediately after those hundred or so entries in the second 4K. In this example, the index symbols will be 13 bits but will be transformed to 12-bit interchange symbols.

8. A program may place the dictionaries in pages that are managed by means of chaining fields at their beginnings. In this case, either the parts of a dictionary have to be moved to be compacted into contiguous locations or there have to be holes in the dictionaries. The definition of COMPRESSION CALL contains nothing explicitly to support holes. However, assuming there is at least one character that never appears in the expanded data, that character can be used as a child character in a parent entry or as a sibling character in a sibling descriptor to specify a child or children that will never be referenced, thus creating a hole.

9. The references to the operands, dictionaries, and symbol-translation table for COMPRESSION CALL may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98.)

10. Figure 7-38 on page 7-81 and Figure 7-39 on page 7-83 show possible forms (not the only possible forms) of the compression and expansion processes. The figures do not show testing for or the results of dictionary errors.
Start
Notes: Entry fields are in parent or sibling descriptor.
SRC=source, DST=destination.

(1) Another SRC char exists? No → Set CC0 and endop.
Yes

(2) Another DST index position exists? No → Set CC1 and endop.
Yes
Use next SRC char as index of alphabet entry.
Call this entry parent.
Advance 1 byte in SRC.

(3) CCT=0? Yes → Store parent index in DST.
No Advance 1 index in DST. → (1)

(4) Set flag=1. Note: Flag is 1 as long as there are equal comparisons to identical leading CCs.

(5) Next SRC char=CC? Yes → Set child index=
No CPTR+CC number (0-origin numbering).
Set flag=0

(6) CCT indicates more children? Yes → ACT=0 OR D=0
No in child? Yes → Call child the parent.
No Advance 1 byte in SRC.

(7) Go to Part 2. Compare SRC chars after next char to AECs in child.

Note: The preferred path for X=0 is shown, but the other path may be taken.

Figure 7-38 (Part 1 of 2). Compression Process
From Part 1. (6)

Set sibling descriptor (SD) index=CPTR+
number of CCs.

(7)

Repeat for each SC in SD.

Note: See the definition for how SCT specifies
number of SCs and more children.

Next SRC char=SC? Yes→Set child index=
SD index+SC number
(1-origin numbering).

Another SC? Yes No

SCT indicates more children? Yes No→(8)

ACT=0 OR D=0 in child? Yes→Call child the parent.
No Advance 1 byte in SRC.

Set SD index =current SD index+number of SCs+1

Note: Second half of format-1 SD is in expansion dictionary.

Enough SRC chars for comparison? No→(8)

Chars equal? Yes→Call child the parent.
No Advance in SRC by 1+number of AEC bytes.
Start

Notes: Fields are in current entry.
SRC=source, DST=destination.

Another SRC index exists? No → Set CC0 and endop.
Yes

Set CC1 and endop.

Another DST byte positions exist? No → Set CC1 and endop.
Yes

Set CC1 and endop.

Next SRC index<256? Yes ───────────/SM59≤≤≤≤Another DST byte
No → Store index as char in DST.

No positions exist? No → Set CC1 and endop.
Yes

Set SYMLEN= PSL+OFST

Get CSL ECs from entry and store in DST.
Advance 1 index in SRC.
Advance 1 byte in DST. →(1)

SYMLEN DST byte positions exist? No → Set CC1 and endop.
Yes

Get PSL ECs from entry and store in DST at OFST.
Use PPTR as index of current entry (see note).

PSL=0? Yes → Get CSL ECs from entry and store in DST.
No

Advance 1 index in SRC.
Advance SYMLEN bytes in DST. →(1)

Get PSL ECs from entry and store in DST at OFST.
Use PPTR as index of current entry (see note).

PSL=0? Yes → Get CSL ECs from entry and store in DST.
No

Advance 1 index in SRC.
Advance SYMLEN bytes in DST. →(1)

Note: If PPTR<256, the action can be:
Store PPTR as char in DST.
Advance 1 index in SRC.
Advance SYMLEN bytes in DST. →(1)

Figure 7-39. Expansion Process
COMPUTE INTERMEDIATE MESSAGE DIGEST (KIMD)

KIMD  R₁,R₂  [RRE]

'8B93E'  ///////////////// R₁  R₂
0   16  24  28  31

COMPUTE LAST MESSAGE DIGEST (KLMD)

KLMD  R₁,R₂  [RRE]

'8B93F'  ///////////////// R₁  R₂
0   16  24  28  31

A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction and the R₁ field are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figures 7-40 and 7-41 show the assigned function codes for COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST, respectively. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for COMPUTE INTERMEDIATE MESSAGE DIGEST are as follows.

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
<th>Parm. Block Size (bytes)</th>
<th>Data Block Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KIMD-Query</td>
<td>16</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>KIMD-SHA-1</td>
<td>20</td>
<td>64</td>
</tr>
</tbody>
</table>

Explanation:
— Not applicable

The function codes for COMPUTE LAST MESSAGE DIGEST are as follows.

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
<th>Parm. Block Size (bytes)</th>
<th>Data Block Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KLMD-Query</td>
<td>16</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>KLMD-SHA-1</td>
<td>28</td>
<td>64</td>
</tr>
</tbody>
</table>

Explanation:
— Not applicable

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R₂ and R₂ + 1 are ignored for the query function.

For all other functions, the second operand is processed as specified by the function code using an initial chaining value in the parameter block, and the result replaces the chaining value. For COMPUTE LAST MESSAGE DIGEST, the operation also uses a message bit length in the parameter block. The operation proceeds until the end of the second-operand location is reached or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The R₂ field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.
The location of the leftmost byte of the second operand is specified by the contents of the $R_2$ general register. The number of bytes in the second-operand location is specified in general register $R_2 + 1$.

As part of the operation, the address in general register $R_2$ is incremented by the number of bytes processed from the second operand, and the length in general register $R_2 + 1$ is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register $R_2$ constitute the address of second operand, and the contents of bit positions 0-39 are ignored; bits 40-63 of the updated address replace the corresponding bits in general register $R_2$, carries out of bit position 40 of the updated address are ignored, and the contents of bit positions 32-39 of general register $R_2$ are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register $R_2$ constitute the address of second operand, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated address replace the corresponding bits in general register $R_2$, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general register $R_2$ is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2$ constitute the address of second operand; bits 0-63 of the updated address replace the contents of general register $R_2$ and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register $R_2 + 1$ form a 32-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of bit positions 32-63 of general register $R_2 + 1$. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register $R_2 + 1$ form a 64-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of general register $R_2 + 1$.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers $R_2$ and $R_2 + 1$, always remain unchanged.

Figure 7-42 on page 7-86 shows the contents of the general registers just described.
In the access-register mode, access registers 1 and R2 specify the address spaces containing the parameter block and second operand, respectively.

The result is obtained as if processing starts at the left end of the second operand and proceeds to the right, block by block. The operation is ended when all source bytes in the second operand have been processed (called normal completion), or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the chaining-value field overlaps any portion of the second operand, the result in the chaining-value field is unpredictable.
For **COMPUTE INTERMEDIATE MESSAGE DIGEST**, normal completion occurs when the number of bytes in the second operand as specified in general register \( R_2 + 1 \) have been processed. For **COMPUTE LAST MESSAGE DIGEST**, after all bytes in the second operand as specified in general register \( R_2 + 1 \) have been processed, the padding operation is performed, and then normal completion occurs.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in \( R_2 + 1 \) is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in \( R_2 + 1 \) is nonzero.

When the second-operand length is initially zero, the second operand is not accessed, general registers \( R_2 \) and \( R_2 + 1 \) are not changed, and condition code 0 is set. For **COMPUTE INTERMEDIATE MESSAGE DIGEST**, the parameter block is not accessed. However, for **COMPUTE LAST MESSAGE DIGEST**, the empty block (\( L = 0 \)) case padding operation is performed and the result is stored into the parameter block.

As observed by other CPUs and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

Access exceptions may be reported for a larger portion of the second operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of the second operand nor for locations more than 4K bytes beyond the current location being processed.

**Symbols Used in Function Descriptions**

The following symbols are used in the subsequent description of the **COMPUTE INTERMEDIATE MESSAGE DIGEST** and **COMPUTE LAST MESSAGE DIGEST** functions. Further description of the secure hash algorithm may be found in *Secure Hash Standard*, Federal Information Processing Standards publication 180-2, National Institute of Standards and Technology, Washington DC, August 1, 2002.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;n&gt; )</td>
<td>Length of item in bytes</td>
</tr>
<tr>
<td>ICV</td>
<td>Initial chaining value</td>
</tr>
<tr>
<td>M</td>
<td>Message block</td>
</tr>
<tr>
<td>OCV</td>
<td>Output chaining value</td>
</tr>
</tbody>
</table>

Figure 7-43. Symbol for SHA-1 Block Digest Algorithm

**KIMD-Query (KIMD Function Code 0)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-42 on page 7-86.

The parameter block used for the function has the following format:

![Figure 7-44. Parameter Block for KIMD-Query](image)

A 128-bit status word is stored in the parameter block. Bits 0-127 of this field correspond to function codes 0-127, respectively, of the **COMPUTE INTERMEDIATE MESSAGE DIGEST** instruction. When a bit is one, the corresponding function is installed; otherwise, the function is not installed.

Condition code 0 is set when execution of the KIMD-Query function completes; condition code 3 is not applicable to this function.

**KIMD-SHA-1 (KIMD Function Code 1)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-42 on page 7-86.

The parameter block used for the function has the following format:
A 20-byte intermediate message digest is generated for the 64-byte message blocks in operand 2 using the SHA-1 block digest algorithm with the 20-byte chaining value in the parameter block. The generated intermediate message digest, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

The message digest for the message (M) in operand 2 is generated using the SHA-1 algorithm with the chaining value and message-bit-length information in the parameter block.

If the length of the message in operand 2 is equal to or greater than 64 bytes, an intermediate message digest is generated for each 64-byte message block using the SHA-1 block digest algorithm with the 20-byte chaining value in the parameter block, and the generated intermediate message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block. This operation is shown in Figure 7-49 on page 7-89 and repeats until the remaining message is less than 64 bytes.

If the length of the message or the remaining message is zero bytes, then the operation in Figure 7-50 on page 7-89 is performed. If the
length of the message or the remaining message is between one byte and 55 bytes inclusive, then the operation in [Figure 7-51 on page 7-89] is performed; if the length is between 56 bytes and 63 bytes inclusive, then the operation in [Figure 7-52 on page 7-90] is performed; The message digest, also called the output chaining value (OCV), is stored into the chaining-value field of the parameter block.

Additional Symbols Used in KLMD Functions

The following additional symbols are used in the description of the COMPUTE LAST MESSAGE DIGEST functions.

Symbol | Explanation for KLMD Function Figures
--- | ---
L | Byte length of operand 2 in storage.
p <n> | n padding bytes; leftmost byte is 80 hex; all other bytes are 00 hex.
z <56> | 56 padding bytes of zero.
mb1 | an 8-byte value specifying the bit length of the total message.
q <64> | a padding block, consisting of 56 bytes of zero followed by an 8-byte mb1.
Special Conditions for KIMD and KLMD

A specification exception is recognized and no other action is taken if any of the following occurs:

1. Bit 56 of general register 0 is not zero.
2. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
3. The $R_2$ field designates an odd-numbered register or general register 0.
4. For COMPUTE INTERMEDIATE MESSAGE DIGEST, the second-operand length is not a multiple of the data block size of the designated function (see Figure 7-40 on page 7-84 to determine the data block sizes for COMPUTE INTERMEDIATE MESSAGE DIGEST functions). This specification-exception condition does not apply to the query function, nor does it apply to COMPUTE LAST MESSAGE DIGEST.

Resulting Condition Code:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Normal completion</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>Partial completion</td>
</tr>
</tbody>
</table>

Program Exceptions:

- Access (fetch, operand 2 and message bit length; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)
- Specification

Figure 7-53. Priority of Execution: KIMD and KLMD

7-90 z/Architecture Principles of Operation
Programming Notes:
1. Bit 56 of general register 0 is reserved for future extension and should be set to zero.
2. When condition code 3 is set, the second operand address and length in general registers R2 and R2 + 1, respectively, and the chaining-value in the parameter block are usually updated such that the program can simply branch back to the instruction to continue the operation.

For unusual situations, the CPU protects against endless reoccurrence for the no-progress case. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop.

3. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3; the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.

4. The instructions COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE LAST MESSAGE DIGEST are designed to be used by a security service application programming interface (API). These APIs provide the program with means to compute the digest of messages of almost unlimited size, including those too large to fit in storage all at once. This is accomplished by permitting the program to pass the message to the API in parts. The following programming notes are described in terms of these APIs.

5. Before processing the first part of a message, the program must set the initial values for the chaining-value field. For SHA-1, the initial chaining values are listed as follows:

   - \( H_0 = \text{'6745 2301'} \)
   - \( H_1 = \text{'EFCD AB89'} \)
   - \( H_2 = \text{'98BA DCFE'} \)
   - \( H_3 = \text{'1325 476'} \)
   - \( H_4 = \text{'C302 E1F0'} \)

6. When processing message parts other than the last, the program must process message parts in multiples of 512 bits (64 bytes) and use the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction.

7. When processing the last message part, the program must compute the length of the original message in bits and place this 64-bit value in the message-bit-length field of the parameter block, and use the COMPUTE LAST MESSAGE DIGEST instruction.

8. The COMPUTE LAST MESSAGE DIGEST instruction does not require the second operand to be a multiple of the block size. It first processes complete blocks, and may set condition code 3 before processing all blocks. After processing all complete blocks, it then performs the padding operation including the remaining portion of the second operand. This may require one or two iterations of the SHA-1 block digest algorithm.

9. The COMPUTE LAST MESSAGE DIGEST instruction provides the SHA-1 padding for messages that are a multiple of eight bits in length. If SHA-1 is to be applied to a bit string which is not a multiple of eight bits, the program must perform the padding and use the COMPUTE INTERMEDIATE MESSAGE DIGEST instruction.

COMPUTE MESSAGE AUTHENTICATION CODE (KMAC)

\[
\begin{array}{c|cccc}
\text{KMAC} & R_1 & R_2 & [RRE] \\
\hline
\text{'B91E'} & / / / / / / & R_1 & R_2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]

A function specified by the function code in general register 0 is performed.

Bits 16-23 of the instruction and the \( R_1 \) field are ignored.

Bit positions 57-63 of general register 0 contain the function code. Figure 7-54 shows the assigned function codes. All other function codes are unassigned. Bit 56 of general register 0 must be zero; otherwise, a specification exception is recognized. All other bits of general register 0 are ignored.

General register 1 contains the logical address of the leftmost byte of the parameter block in storage. In the 24-bit addressing mode, the contents of bit positions 40-63 of general register 1
constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register 1 constitute the address.

The function codes for COMPUTE MESSAGE AUTHENTICATION CODE are as follows.

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
<th>Parm. Block Size (bytes)</th>
<th>Data Block Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KMAC-Query</td>
<td>16</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>KMAC-DEA</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>KMAC-TDEA-128</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>KMAC-TDEA-192</td>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

**Explanation:**
— Not applicable

All other function codes are unassigned.

The query function provides the means of indicating the availability of the other functions. The contents of general registers R\_2 and R\_2 + 1 are ignored.

For all other functions, the second operand is processed as specified by the function code using an initial chaining value in the parameter block and the result replaces the chaining value. The operation also uses a cryptographic key in the parameter block. The operation proceeds until the end of the second-operand location is reached or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The R\_2 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the second operand is specified by the contents of the R\_2 general register. The number of bytes in the second-operand location is specified in general register R\_2 + 1.

As part of the operation, the address in general register R\_2 is incremented by the number of bytes processed from the second operand, and the length in general register R\_2 + 1 is decremented by the same number. The formation and updating of the address and length is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general register R\_2 constitute the address of second operand, and are ignored; bits 40-63 of the updated address replace the corresponding bits in general register R\_2, carries out of bit position 40 of the updated address are ignored and, the contents of bit positions 32-39 of general register R\_2 are set to zeros. In the 31-bit addressing mode, the contents of bit positions 33-63 of general register R\_2 constitute the address of second operand, and the contents of bit positions 0-32 are ignored; bits 33-63 of the updated address replace the corresponding bits in general register R\_2, carries out of bit position 33 of the updated address are ignored, and the content of bit position 32 of general register R\_2 is set to zero. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R\_2 constitute the address of second operand; bits 0-63 of the updated address replace the contents of general register R\_2 and carries out of bit position 0 are ignored.

In both the 24-bit and the 31-bit addressing modes, the contents of bit positions 32-63 of general register R\_2 + 1 form a 32-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of bit positions 32-63 of general register R\_2 + 1. In the 64-bit addressing mode, the contents of bit positions 0-63 of general register R\_2 + 1 form a 64-bit unsigned binary integer which specifies the number of bytes in the second operand; and the updated value replaces the contents of general register R\_2 + 1.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R\_2 and R\_2 + 1, always remain unchanged.

Figure 7-54 on page 7-93 shows the contents of the general registers just described.

Figure 7-55 on page 7-93 shows the contents of the general registers just described.
In the access-register mode, access registers 1 and R2 specify the address spaces containing the parameter block and second operand, respectively.

The result is obtained as if processing starts at the left end of the second operand and proceeds to the right, block by block. The operation is ended when all source bytes in the second operand have been processed (called normal completion), or when a CPU-determined number of blocks that is less than the length of the second operand have been processed (called partial completion). The CPU-determined number of blocks depends on the model, and may be a different number each time the instruction is executed. The CPU-determined number of blocks is usually nonzero. In certain unusual situations, this number may be zero, and condition code 3 may be set with no progress. However, the CPU protects against endless reoccurrence of this no-progress case.

When the chaining-value field overlaps any portion of the second operand, the result in the chaining-value field is unpredictable.
Normal completion occurs when the number of bytes in the second operand as specified in general register R$_2 + 1$ have been processed.

When the operation ends due to normal completion, condition code 0 is set and the resulting value in R$_2 + 1$ is zero. When the operation ends due to partial completion, condition code 3 is set and the resulting value in R$_2 + 1$ is nonzero.

When the second-operand length is initially zero, the second operand and the parameter block are not accessed, general registers R$_2$ and R$_2 + 1$ are not changed, and condition code 0 is set.

As observed by other CPUs and channel programs, references to the parameter block and storage operands may be multiple-access references, accesses to these storage locations are not necessarily block-concurrent, and the sequence of these accesses or references is undefined.

Access exceptions may be reported for a larger portion of the second operand than is processed in a single execution of the instruction; however, access exceptions are not recognized for locations beyond the length of the second operand nor for locations more than 4K bytes beyond the current location being processed.

**Symbols Used in Function Descriptions**

The following symbols are used in the subsequent description of the COMPUTE MESSAGE AUTHENTICATION CODE functions. For data-encryption-algorithm (DEA) functions, the DEA-key-parity bit in each byte of the DEA key is ignored, and the operation proceeds normally, regardless of the DEA-key parity of the key. Further description of the data-encryption algorithm may be found in *Data Encryption Algorithm*, ANSI-X3.92.1981, American National Standard for Information Systems.

Circle the text for potential errors.
The message authentication code for the 8-byte message blocks (M1, M2, ..., Mn) in operand 2 is computed using the DEA algorithm with the 64-bit cryptographic key and the 64-bit chaining value in the parameter block.

The message authentication code, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

Figure 7-60. KMAC-DEA

**KMAC-TDEA-128 (Function Code 2)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-55 on page 7-93.

The parameter block used for the function has the following format:

<table>
<thead>
<tr>
<th></th>
<th>Chaining Value (CV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Cryptographic Key 1 (K1)</td>
</tr>
<tr>
<td>16</td>
<td>Cryptographic Key 2 (K2)</td>
</tr>
</tbody>
</table>

Figure 7-61. Parameter Block for KMAC-TDEA-128

The message authentication code for the 8-byte message blocks (M1, M2, ..., Mn) in operand 2 is computed using the TDEA algorithm with the two 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.

The message authentication code, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

Figure 7-62. KMAC-TDEA-128

**KMAC-TDEA-192 (Function Code 3)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-55 on page 7-93.

The parameter block used for the function has the following format:

<table>
<thead>
<tr>
<th></th>
<th>Chaining Value (CV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Cryptographic Key 1 (K1)</td>
</tr>
<tr>
<td>16</td>
<td>Cryptographic Key 2 (K2)</td>
</tr>
<tr>
<td>24</td>
<td>Cryptographic Key 3 (K3)</td>
</tr>
</tbody>
</table>

Figure 7-63. Parameter Block for KMAC-TDEA-192

The message authentication code for the 8-byte message blocks (M1, M2, ..., Mn) in operand 2 is computed using the TDEA algorithm with the three 64-bit cryptographic keys and the 64-bit chaining value in the parameter block.
The message authentication code, also called the output chaining value (OCV), is stored in the chaining-value field of the parameter block. The operation is shown in the following figure:

![Diagram](image)

**Figure 7-64. KMAC-TDEA-192**

**Special Conditions for KMAC**

1. Exceptions with the same priority as the priority of program-interruption conditions for the general case.
3. Operation exception.
4. Specification exception due to invalid function code or invalid register number.
5. Specification exception due to invalid operand length.
6. Condition code 0 due to second-operand length originally zero.
7. Access exceptions for an access to the parameter block or second operand.
8. Condition code 0 due to normal completion (second-operand length originally nonzero, but stepped to zero).
9. Condition code 3 due to partial completion (second-operand length still nonzero).

**Resulting Condition Code:**
0 Normal completion
1 --
2 --
3 Partial completion

**Program Exceptions:**
- Access (fetch, operand 2, cryptographic key; fetch and store, chaining value)
- Operation (if the message-security assist is not installed)
- Specification

A specification exception is recognized and no other action is taken if any of the following occurs:
1. Bit 56 of general register 0 is not zero.
2. Bits 57-63 of general register 0 specify an unassigned or uninstalled function code.
3. The R2 field designates an odd-numbered register or general register 0.
4. The second-operand length is not a multiple of the data block size of the designated function (see Figure 7-54 on page 7-92 to determine the data block size for COMPUTE MESSAGE AUTHENTICATION CODE functions).

**Figure 7-65. Priority of Execution: KMAC**
Programming Notes:
1. Bit 56 of general register 0 is reserved for future extension and should be set to zero.
2. When condition code 3 is set, the second operand address and length in general registers $R_2$ and $R_2 + 1$, respectively, and the chaining-value in the parameter block are usually updated such that the program can simply branch back to the instruction to continue the operation. For unusual situations, the CPU protects against endless recurrence for the no-progress case. Thus, the program can safely branch back to the instruction whenever condition code 3 is set with no exposure to an endless loop.
3. If the length of the second operand is nonzero initially and condition code 0 is set, the registers are updated in the same manner as for condition code 3; the chaining value in this case is such that additional operands can be processed as if they were part of the same chain.
4. Before processing the first part of a message, the program must set the initial values for the chaining-value field. To comply with ANSI X9.9 or X9.19, the initial chaining value shall be set to all binary zeros.

CONVERT TO BINARY

FOR CONVERT TO BINARY (CVB, CVBY), the second operand occupies eight bytes in storage, and, for CONVERT TO BINARY (CVBG), the second operand occupies sixteen bytes in storage. The second operand has the format of packed decimal data, as described in [Chapter 8, “Decimal Instructions.”] It is checked for valid sign and digit codes, and a decimal-operand data exception is recognized when an invalid code is detected.

For CONVERT TO BINARY (CVB, CVBY), the result of the conversion is a 32-bit signed binary integer, which is placed in bit positions 32-63 of general register $R_1$. Bits 0-31 of the register remain unchanged. The maximum positive number that can be converted and still be contained in 32 bit positions is $2,147,483,647$; the maximum negative number (the negative number with the greatest absolute value) that can be converted is $-2,147,483,648$. For any decimal number outside this range, the operation is completed by placing the 32 rightmost bits of the binary result in the register, and a fixed-point-divide exception is recognized.

For CONVERT TO BINARY (CVBG), the result of the conversion is a 64-bit signed binary integer, which is placed in bit positions 0-63 of general register $R_1$. The maximum positive number that can be converted and still be contained in a 64-bit register is $9,223,372,036,854,775,807$; the maximum negative number (the negative number with the greatest absolute value) that can be converted is $-9,223,372,036,854,775,808$. For any decimal number outside this range, a fixed-point-divide exception is recognized, and the operation is suppressed.

The displacement for CVB is treated as a 12-bit unsigned binary integer. The displacement for CVBY and CVBG is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2)
- Data
- Fixed-point divide
- Operation (CVBY, if the long-displacement facility is not installed)
Programming Notes:

1. An example of the use of the CONVERT TO BINARY instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. When the second operand is negative, the result is in two's-complement notation.

3. The storage-operand references for CONVERT TO BINARY may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98.)

## CONVERT TO DECIMAL

**CVD** \( R_1, D_2(X_2, B_2) \) \[ RX \]

<table>
<thead>
<tr>
<th>'4E'</th>
<th>( R_1 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( D_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
\text{CVDY} \quad \begin{array}{cccccc}
'\text{E3}' & \( R_1 \) & \( X_2 \) & \( B_2 \) & \( DL_2 \) & \( DH_2 \) & '26'
\end{array}
\]

| 0 | 8 | 12 | 16 | 20 | 32 | 40 | 47 |

**CVDG** \( R_1, D_2(X_2, B_2) \) \[ RXY \]

| 'E3' | \( R_1 \) | \( X_2 \) | \( B_2 \) | \( DL_2 \) | \( DH_2 \) | '2E'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
</tr>
</tbody>
</table>

The first operand is changed from binary to decimal, and the result is stored at the second-operand location.

For CONVERT TO DECIMAL (CVD, CVDY), the first operand is treated as a 32-bit signed binary integer, and the result occupies eight bytes in storage. For CONVERT TO DECIMAL (CVDG), the first operand is treated as a 64-bit signed binary integer, and the result occupies sixteen bytes in storage.

The result is in the format for packed decimal data, as described in Chapter 8, "Decimal Instructions." The rightmost four bits of the result represent the sign. A positive sign is encoded as 1100; a negative sign is encoded as 1101.

The displacement for CVD is treated as a 12-bit unsigned binary integer. The displacement for CVDY and CVDG is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Operation (CVDY, if the long-displacement facility is not installed)

Programming Notes:

1. An example of the use of the CONVERT TO DECIMAL instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. For CVD and CVDY, the number to be converted is a 32-bit signed binary integer obtained from a general register. Since 15 decimal digits are available for the result, and the decimal equivalent of 31 bits requires at most 10 decimal digits, an overflow cannot occur. Similarly, for CVDG, 31 decimal digits are available, the decimal equivalent of 63 bits is at most 19 digits, and an overflow cannot occur.

3. The storage-operand references for CONVERT TO DECIMAL may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98.)

## CONVERT UTF-16 TO UTF-32

**CU24** \( R_1, R_2 \) \[ RRE \]

<table>
<thead>
<tr>
<th>'B9B1'</th>
<th>( \text{/////} )</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The two-byte UTF-16 (Unicode) characters of the second operand are converted to UTF-32 characters and placed at the first-operand location. The UTF-32 characters are four bytes. The operation proceeds until the end of the first or second operand is reached or a CPU-determined number of characters have been converted, whichever occurs first. The result is indicated in the condition code.

The \( R_1 \) and \( R_2 \) fields each designate an even-odd pair of general registers and must designate an
even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R₁ and R₂, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers R₁ + 1 and R₂ + 1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers R₁ + 1 and R₂ + 1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R₁ and R₂ is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R₁ and R₂ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-66 on page 7-100.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The characters resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted or a CPU-determined number of remaining bytes is converted.

To show the method of converting a UTF-16 character to a UTF-32 character, the bits of a Unicode character are identified by letters as follows:

Unicode Character 111111
Bit Numbers 01234567 89012345
Identifying Bit Letters abcdedefgh ijklmnop

In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

Unicode High Surrogate

Bit Numbers 01234567 89012345
Identifying Bit Letters abcdedefgh ijklmnop

Unicode Low Surrogate

Bit Numbers 67890123 45678901
Identifying Bit Letters mnopqrstuvwxyz

Any Unicode character in the range 0000 to D7FF and DC00 to FFFF hex is converted to a four-byte UTF-32 character as follows:

Unicode 111111
Character

UTF-32 00000000 00000000 abcdedefgh ijklmnop
Character

where uvwxy = abcd + 1

The first six bits of the second Unicode character are ignored.

The second-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes when the first two bytes are a Unicode high surrogate. The first-operand location is considered exhausted when it does not contain at least four remaining bytes.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been converted, condition code 3 is set.

When the operation is completed, the contents of general register R₂ + 1 are decremented by the number of bytes converted, and the contents of general register R₁ are incremented by the same number. Also, the contents of general register R₁ + 1 are decremented by the number of bytes placed at the first-operand location, and the con-
24-Bit Addressing Mode

<table>
<thead>
<tr>
<th>R1</th>
<th>First-Operand Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-40</td>
<td>63</td>
</tr>
<tr>
<td>R1 + 1</td>
<td>First-Operand Length</td>
</tr>
<tr>
<td>0-32</td>
<td>63</td>
</tr>
<tr>
<td>R2</td>
<td>Second-Operand Address</td>
</tr>
<tr>
<td>0-40</td>
<td>63</td>
</tr>
<tr>
<td>R2 + 1</td>
<td>Second-Operand Length</td>
</tr>
<tr>
<td>0-32</td>
<td>63</td>
</tr>
</tbody>
</table>

31-Bit Addressing Mode

<table>
<thead>
<tr>
<th>R1</th>
<th>First-Operand Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-33</td>
<td>63</td>
</tr>
<tr>
<td>R1 + 1</td>
<td>First-Operand Length</td>
</tr>
<tr>
<td>0-32</td>
<td>63</td>
</tr>
<tr>
<td>R2</td>
<td>Second-Operand Address</td>
</tr>
<tr>
<td>0-33</td>
<td>63</td>
</tr>
<tr>
<td>R2 + 1</td>
<td>Second-Operand Length</td>
</tr>
<tr>
<td>0-32</td>
<td>63</td>
</tr>
</tbody>
</table>

64-Bit Addressing Mode

<table>
<thead>
<tr>
<th>R1</th>
<th>First-Operand Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>R1 + 1</td>
<td>First-Operand Length</td>
</tr>
<tr>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>R2</td>
<td>Second-Operand Address</td>
</tr>
<tr>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>R2 + 1</td>
<td>Second-Operand Length</td>
</tr>
<tr>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

Figure 7-66. Register Contents for CONVERT UTF-16 TO UTF-32

- The contents of general register R1 are incremented by the same number. When general registers R1 and R2 are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

- In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R1, R1 + 1, R2, and R2 + 1, always remain unchanged.

- When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

- The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.
When the $R_1$ register is the same register as the $R_2$ register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the $R$ field associated with that operand is odd.

**Resulting Condition Code:**
- 0 Entire second operand processed
- 1 End of first operand reached
- 2 --
- 3 CPU-determined number of characters converted

**Program Exceptions:**
- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification

**Programming Notes:**
1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.
2. The storage-operand references of CONVERT UTF-16 TO UTF-32 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98)

### CONVERT UTF-16 TO UTF-8

<table>
<thead>
<tr>
<th>CU21</th>
<th>$R_1$, $R_2$ [RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUUTF</td>
<td>$R_1$, $R_2$ [RRE]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>'B2A6'</th>
<th>// //////</th>
<th>$R_1$</th>
<th>$R_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The two-byte Unicode characters of the second operand are converted to UTF-8 characters and placed at the first-operand location. The UTF-8 characters are one, two, three, or four bytes, depending on the Unicode characters that are converted. The operation proceeds until the end of the first or second operand is reached or a CPU-determined number of characters have been converted, whichever occurs first. The result is indicated in the condition code.

The $R_1$ and $R_2$ fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers $R_1$ and $R_2$, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers $R_1 + 1$ and $R_2 + 1$, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers $R_1$ and $R_2$ is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers $R_1$ and $R_2$ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In
the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-67.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted or a CPU-determined number of second-operand characters have been converted.

To show the method of converting a Unicode character to a UTF-8 character, the bits of a Unicode character are identified by letters as follows:

<table>
<thead>
<tr>
<th>Unicode Character</th>
<th>111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Numbers</td>
<td>01234567 89012345</td>
</tr>
<tr>
<td>Identifying Bit Letters</td>
<td>abcdefgh ijklnmop</td>
</tr>
</tbody>
</table>
In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

Unicode High Surrogate: 111111

Bit Numbers: 01234567 89012345

Identifying Bit Letters: abcd efgh ijkl

Unicode Low Surrogate: 111222 22222233

Bit Numbers: 67890123 45678901

Identifying Bit Letters: 111111kl mnopqrst

Any Unicode character in the range 0000 to 007F hex is converted to a one-byte UTF-8 character as follows:

Unicode Character: 00000000 0jklmnop

UTF-8 Character: 0jklmnop

Any Unicode character in the range 0080 to 07FF hex is converted to a two-byte UTF-8 character as follows:

Unicode Character: 0000fgh ijklnmop

UTF-8 Character: 110fghij 10klmnop

Any Unicode character in the range 0800 to D7FF and DC00 to FFFF hex is converted to a three-byte UTF-8 character as follows:

Unicode Character: abcdefgh ijklnmop

UTF-8 Character: 1110abcd 10efghij 10klmnop

Any Unicode surrogate pair starting with a high surrogate in the range D800 to DBFF hex is converted to a four-byte UTF-8 character as follows:

Unicode Characters: 110110ab cdefghij 110111kl mnopqrstuvwxyz

UTF-8 Character: 11110uvw 10xyefgh 10ijklmn 10opqrstuvwxyz

where uvwxy = abcd + 1

The first six bits of the second Unicode character are ignored.

The second-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes when the first two bytes are a Unicode high surrogate. The first-operand location is considered exhausted when it does not contain at least the one, two, three, or four remaining bytes required to contain the UTF-8 character resulting from the conversion of the next second-operand character or surrogate pair.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been converted, condition code 3 is set.

When the operation is completed, the contents of general register $R_1 + 1$ are decremented by the number of bytes converted, and the contents of general register $R_2$ are incremented by the same number. Also, the contents of general register $R_3$ are decremented by the number of bytes placed at the first-operand location, and the contents of general register $R_3$ are incremented by the same number. When general registers $R_1$ and $R_2$ are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers $R_1$, $R_1 + 1$, $R_3$, and $R_3 + 1$, always remain unchanged.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the $R_1$ register is the same register as the $R_2$ register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recog-
nized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

**Resulting Condition Code:**

0  Entire second operand processed
1  End of first operand reached
2  --
3  CPU-determined number of characters converted

**Program Exceptions:**

- Access (fetch, operand 2; store, operand 1)
- Specification

**Programming Notes:**

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.

2. The storage-operand references of CONVERT UNICODE TO UTF-8 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98)

**CONVERT UTF-32 TO UTF-16**

CU42 R₁,R₂ [RRE]

<table>
<thead>
<tr>
<th>'B9B3'</th>
<th>//////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The four-byte UTF-32 characters of the second operand are converted to two-byte UTF-16 characters and placed at the first-operand location. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid UTF-32 character is encountered, whichever occurs first. The result is indicated in the condition code.

The R₁ and R₂ fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R₁ and R₂, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers R₁ + 1 and R₂ + 1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers R₁ + 1 and R₂ + 1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R₁ and R₂ is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R₁ and R₂ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-68 on page 7-105.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or an invalid UTF-32 character is encountered in the second operand.

To show the method of converting a UTF-32 character to a UTF-16 character, the bits of a UTF-32 character in the range 00000000 to 0000D7FF and 0000DC00 to 0000FFFF hex are identified by letters as follows:
The bits of a UTF-32 character in the range 00010000 to 0010FFFF hex are identified by letters as follows:

The bits of a UTF-16 character in the range 0000 to D7FF and DC00 to FFFF hex are identified by letters as follows:
In the case of a UTF-16 surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

<table>
<thead>
<tr>
<th>UTF-32 High Surrogate</th>
<th>111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Numbers</td>
<td>01234567 89012345</td>
</tr>
<tr>
<td>Identifying Bit Letters</td>
<td>110110ab cdefghij</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UTF-32 Low Surrogate</th>
<th>11112222 22222223</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Numbers</td>
<td>67890123 45678901</td>
</tr>
<tr>
<td>Identifying Bit Letters</td>
<td>110111kl mnopqrstuvwxyz</td>
</tr>
</tbody>
</table>

When the contents of the UTF-32 character are in the range 00000000 to 0000D7FF and 0000DC00 to 0000FFFF hex, the character is converted to a two-byte UTF-32 character as follows:

<table>
<thead>
<tr>
<th>UTF-32</th>
<th>00000000 00000000 abcdefgh iklm nop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UTF-16</th>
<th>abcdefgh iklmnop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td></td>
</tr>
</tbody>
</table>

When the contents of the UTF-32 character are in the range 00010000 to 0010FFFF hex, the character is converted to two two-byte UTF-16 characters (a surrogate pair) as follows:

<table>
<thead>
<tr>
<th>UTF-32</th>
<th>00000000 000uvwxy efg hijkl mnopqrstuvwxyz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UTF-16</th>
<th>110110ab cdefghij 110111kl mnopqrstuvwxyz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characters</td>
<td></td>
</tr>
</tbody>
</table>

where zabcd = uvwxy -1

The high order bit (z) produced by the subtract operation should be zero but is ignored.

The second-operand location is considered exhausted when it does not contain at least four remaining bytes. The first-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes in the case when a UTF-32 character are in the range 00010000 to 0010FFFF hex is converted.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been processed, condition code 3 is set.

When the contents of the next UTF-32 character is in the range 0000D800 to 0000DBFF or 00110000 to FFFFFFFF hex, the character is invalid, and condition code 2 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register R₂ + 1 are decremented by the number of bytes converted, and the contents of general register R₂ are incremented by the same number. Also, the contents of general register R₁ + 1 are decremented by the number of bytes placed at the first-operand location, and the contents of general register R₁ are incremented by the same number. When general registers R₁ and R₂ are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R₁, R₁ + 1, R₂, and R₂ + 1, always remain unchanged.

When condition code 2 is set, general register R₂ contains the address of the invalid UTF-32 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R₁ register is the same register as the R₂ register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.
When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

**Resulting Condition Code:**
- 0: Entire second operand processed
- 1: End of first operand reached
- 2: Invalid UTF-32 character
- 3: CPU-determined number of characters processed

**Program Exceptions:**
- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification

**Programming Notes:**
1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.
2. The storage-operand references of CONVERT UTF-32 TO UTF-16 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98)

**CONVERT UTF-32 TO UTF-8**

```c
CU41 R1,R2 [RRE]
```

<table>
<thead>
<tr>
<th>'B9B2'</th>
<th>/ / / / / /</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The four-byte UTF-32 characters of the second operand are converted to UTF-8 characters and placed at the first-operand location. The UTF-8 characters are one, two, three, or four bytes, depending on the Unicode characters that are converted. The operation proceeds until the end of the first or second operand is reached or a CPU-determined number of characters have been converted, whichever occurs first. The result is indicated in the condition code.

The R1 and R2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R1 and R2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers R1 + 1 and R2 + 1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers R1 + 1 and R2 + 1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R1 and R2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R1 and R2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-69 on page 7-108.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted or a CPU-determined number of second-operand characters have been converted.

To show the method of converting a UTF-32 character to a UTF-8 character, the bits of a UTF-32 character in the range 00000000 to 0000D7FF and 0000DC00 to 0000FFFF hex are identified by letters as follows:

```
<table>
<thead>
<tr>
<th>UTF-32</th>
<th>111111 11112222 22222233</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>01234567 89012345 67890123 45678901</td>
</tr>
<tr>
<td>Bit Numbers</td>
<td>Identifying 00000000 00000000 abcd efgh ijkl mn op</td>
</tr>
</tbody>
</table>
```

Chapter 7. General Instructions 7-107
The bits of a UTF-32 character in the range 00010000 to 0010FFFF hex (UTF-16 surrogate pair) are identified by letters as follows:

| UTF-32 | 00000000 00000000 00000000 00000000 0jklmnop Character
| UTF-8  | 0jklmnop Character

Any UTF-32 character in the range 00000080 to 000007FF hex is converted to a two-byte UTF-8 character as follows:

| UTF-32 | 00000000 00000000 00000000 00000000 000000fgh ijklmnop Character
| UTF-8  | 00000000 00000000 00000000 00000000 000000fgh ijklmnop Character

Any UTF-32 character in the range 00000000 to 0000007F hex is converted to a one-byte UTF-8 character as follows:

| UTF-32 | 00000000 00000000 00000000 00000000 010fghij 10klmnop Character
| UTF-8  | 010fghij 10klmnop Character
Any UTF-32 character in the range 00000800 to 0000D7FF and 0000DC00 to 0000FFFF hex is converted to a three-byte UTF-8 character as follows:

```
UTF-32   00000000 00000000 abcdefgh ijklmnop
    Character
UTF-8    1110abcd 10efghij 10klmnop
    Character
```

Any UTF-32 character in the range 00010000 to 0010FFFF hex is converted to a four-byte UTF-8 character as follows:

```
UTF-32   00000000 000uvwxy efghijkl mnopqrst
    Characters
UTF-8    1111uvw 10xyefgh 10ijklmn 10opqrst
    Character
```

The second-operand location is considered exhausted when it does not contain at least four remaining bytes. The first-operand location is considered exhausted when it does not contain at least the one, two, three, or four remaining bytes required to contain the UTF-8 character resulting from the conversion of the next second-operand character or surrogate pair.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been converted, condition code 3 is set.

When the UTF-32 character is in the range 0000D800 and 0000DBFF or 00110000 to FFFFFFFF hex, the character is invalid, and condition code 2 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register R₂ + 1 are decremented by the number of bytes converted, and the contents of general register R₂ are incremented by the same number. Also, the contents of general register R₁ + 1 are decremented by the number of bytes placed at the first-operand location, and the contents of general register R₁ are incremented by the same number. When general registers R₁ and R₂ are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R₁, R₁ + 1, R₂, and R₂ + 1, always remain unchanged.

When condition code 2 is set, general register R₂ contains the address of the invalid UTF-32 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R₁ register is the same register as the R₂ register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the R field associated with that operand is odd.

**Resulting Condition Code:**

- 0  Entire second operand processed
- 1  End of first operand reached
- 2  Invalid UTF-32 character.
- 3  CPU-determined number of characters converted

**Program Exceptions:**

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification
Programming Notes:
1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.

2. The storage-operand references of CONVERT UTF-32 TO UTF-8 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98.)

CONVERT UTF-8 TO UTF-16

CONVERT UTF-8 TO UNICODE

CU12 R1.R2 [RRE]
CUTFU R1, R2 [RRE]

<table>
<thead>
<tr>
<th>'B2A7'</th>
<th>////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The one-, two-, three-, or four-byte UTF-8 characters of the second operand are converted to two-byte Unicode characters and placed at the first-operand location. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid UTF-8 character is encountered, whichever occurs first. The result is indicated in the condition code.

The R1 and R2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R1 and R2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers R1 + 1 and R2 + 1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers R1 + 1 and R2 + 1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R1 and R2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R1 and R2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-70 on page 7-111.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or an invalid UTF-8 character is encountered in the second operand.

To show the method of converting a UTF-8 character to a Unicode character, the bits of a Unicode character are identified by letters as follows:

Unicode Character 111111
Bit Numbers 01234567 89012345
Identifying Bit Letters abcdefgh ijklnmop

In the case of a Unicode surrogate pair, which is a character pair consisting of a character called a high surrogate followed by a character called a low surrogate, the bits are identified by letters as follows:

Unicode High Surrogate 111111
Bit Numbers 01234567 89012345
Identifying Bit Letters 110110ab cdefghij

Unicode Low Surrogate 11112222 22222233
Bit Numbers 67890123 45678901
Identifying Bit Letters 110111kl mnopqrstuvwxyz

When the contents of the first byte of a UTF-8 character are in the range 00 to 7F hex, the character is a one-byte character, and it is converted to a two-byte Unicode character as follows:
When the contents of the first byte of a UTF-8 character are in the range C0 to DF hex, the character is a two-byte character, and it is converted to a two-byte Unicode character as follows:

The first two bits in the second byte of the UTF-8 character are ignored.

When the contents of the first byte of a UTF-8 character are in the range E0 to EF hex, the character is a three-byte character, and it is converted to a two-byte Unicode character as follows:
The first two bits in the second and third bytes of the UTF-8 character are ignored.

When the contents of the first byte of a UTF-8 character are in the range F0 to F7 hex, the character is a four-byte character, and it is converted to two two-byte Unicode characters (a surrogate pair) as follows:

UTF-8 \[\text{1111} / \text{uxyyyy} \text{10ijklmm} \text{10opqrstuvwxyz}\]

Unicode \[\text{110110ab cdefghij 110111kl mnpqrstuvwxyz}\]

where \(\text{abcd} = \text{uvwxyz} - 1\)

The first two bits in the second, third, and fourth bytes of the UTF-8 character are ignored. The high order bit (2) produced by the subtract operation should be zero but is ignored.

The second-operand location is considered exhausted when it does not contain at least one remaining byte or when it does not contain at least the two, three, or four remaining bytes required to contain the two-, three-, or four-byte UTF-8 character indicated by the contents of the first remaining byte. The first-operand location is considered exhausted when it does not contain at least two remaining bytes or at least four remaining bytes in the case when a four byte UTF-8 character is to be converted.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been processed, condition code 3 is set.

When the contents of the first byte of the next UTF-8 character are in the range 80 to BF hex or F8 to FF hex, the character is invalid, and condition code 2 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register \(R_2\) are incremented by the same number. Also, the contents of general register \(R_1 + 1\) are decremented by the number of bytes placed at the first-operand location, and the contents of general register \(R_1\) are incremented by the same number. When general registers \(R_1\) and \(R_2\) are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers \(R_1\), \(R_1 + 1\), \(R_2\), and \(R_2 + 1\), always remain unchanged.

When condition code 2 is set, general register \(R_2\) contains the address of the invalid UTF-8 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the \(R_1\) register is the same register as the \(R_2\) register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand. Access exceptions are not recognized for an operand if the \(R\) field associated with that operand is odd.

Resulting Condition Code:

- 0: Entire second operand processed
- 1: End of first operand reached
- 2: Invalid UTF-8 character
3 CPU-determined number of characters processed

Program Exceptions:
- Access (fetch, operand 2; store, operand 1)
- Specification

Programming Notes:
1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.

2. Bits 0 and 1 of the continuation bytes of multiple-byte UTF-8 characters are not checked in order to improve the performance of the conversion. Therefore, invalid continuation bytes are not detected.

3. The storage-operand references of CONVERT UTF-8 TO UTF-16 may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

CONVERT UTF-8 TO UTF-32

CU14 R1,R2 [RRE]

<table>
<thead>
<tr>
<th>'B9B0'</th>
<th>////////////// R1 R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
</tr>
</tbody>
</table>

The one-, two-, three-, or four-byte UTF-8 characters of the second operand are converted to four-byte UTF-32 characters and placed at the first-operand location. The operation proceeds until the end of the first or second operand is reached, a CPU-determined number of characters have been converted, or an invalid UTF-8 character is encountered, whichever occurs first. The result is indicated in the condition code.

The R1 and R2 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and the second operand is designated by the contents of general registers R1 and R2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the contents of bit positions 32-63 of general registers R1 + 1 and R2 + 1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and second-operand locations is specified by the entire contents of general registers R1 + 1 and R2 + 1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R1 and R2 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R1 and R2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-71 on page 7-114.

The characters of the second operand are selected one by one for conversion, proceeding left to right. The bytes resulting from a conversion are placed at the first-operand location, proceeding left to right. The operation proceeds until the first-operand or second-operand location is exhausted, a CPU-determined number of second-operand characters have been converted, or an invalid UTF-8 character is encountered in the second operand.

To show the method of converting a UTF-8 character to a UTF-32 character, the bits of a UTF-32 character in the range 00000000 0000D7FF and 0000DC00 to 0000FFFF hex are identified by letters as follows:

<table>
<thead>
<tr>
<th>UTF-32</th>
<th>Character</th>
<th>111111 11112222 22222233</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01234567</td>
<td>89012345 67890123 45678901</td>
</tr>
<tr>
<td>Bit Numbers</td>
<td>Identifying 00000000 00000000 abcdefgh ijklnmop</td>
<td></td>
</tr>
</tbody>
</table>

the bits of a UTF-32 character in the range 00010000 0010FFFF hex (UTF-16 surrogate pair) are identified by letters as follows:
### 24-Bit Addressing Mode

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0-40</td>
<td>First-Operand Address</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;1&lt;/sub&gt; + 1</td>
<td>0-32</td>
<td>First-Operand Length</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;2&lt;/sub&gt;</td>
<td>0-40</td>
<td>Second-Operand Address</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;2&lt;/sub&gt; + 1</td>
<td>0-32</td>
<td>Second-Operand Length</td>
<td>0-63</td>
</tr>
</tbody>
</table>

### 31-Bit Addressing Mode

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0-33</td>
<td>First-Operand Address</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;1&lt;/sub&gt; + 1</td>
<td>0-32</td>
<td>First-Operand Length</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;2&lt;/sub&gt;</td>
<td>0-33</td>
<td>Second-Operand Address</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;2&lt;/sub&gt; + 1</td>
<td>0-32</td>
<td>Second-Operand Length</td>
<td>0-63</td>
</tr>
</tbody>
</table>

### 64-Bit Addressing Mode

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0-63</td>
<td>First-Operand Address</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;1&lt;/sub&gt; + 1</td>
<td>0-63</td>
<td>First-Operand Length</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;2&lt;/sub&gt;</td>
<td>0-63</td>
<td>Second-Operand Address</td>
<td>0-63</td>
</tr>
<tr>
<td>R&lt;sub&gt;2&lt;/sub&gt; + 1</td>
<td>0-63</td>
<td>Second-Operand Length</td>
<td>0-63</td>
</tr>
</tbody>
</table>

#### Figure 7-71. Register Contents for CONVERT UTF-8 TO UTF32

<table>
<thead>
<tr>
<th>UTF-32 Character</th>
<th>111111 11112222 22222233</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Numbers</td>
<td>01234567 89012345 67890123 45678901</td>
</tr>
<tr>
<td>Identifying</td>
<td>00000000 0000vwxy efgijklnopqrstuvwxyz</td>
</tr>
</tbody>
</table>

When the contents of the first byte of a UTF-8 character are in the range 00 to 7F hex, the character is a one-byte character, and it is converted to a four-byte UTF-32 character as follows:

When the contents of the first byte of a UTF-8 character are in the range C0 to DF hex, the character is a two-byte character, and it is converted to a four-byte UTF-32 character as follows:
The first two bits in the second byte of the UTF-8 character are ignored.

When the contents of the first byte of a UTF-8 character are in the range E0 to EF hex, the character is a three-byte character, and it is converted to a four-byte UTF-32 character as follows:

```
| UTF-8 | 110fghij 10klmnop |
| utf32 | 00000000 00000000 00000000 fgh ijklmnop |
```

The first two bits in the second and third bytes of the UTF-8 character are ignored.

When the contents of the first byte of a UTF-8 character are in the range F0 to F7 hex, the character is a four-byte character, and it is converted to a four-byte UTF-32 character (a surrogate pair) as follows:

```
| UTF-8 | 1110abcd 10efghij 10klmnop |
| utf32 | 00000000 00000000 abcd efghij klmnop |
```

The first two bits in the second, third, and fourth bytes of the UTF-8 character are ignored.

The second-operand location is considered exhausted when it does not contain at least one remaining byte or when it does not contain at least the two, three, or four remaining bytes required to contain the two-, three-, or four-byte UTF-8 character indicated by the contents of the first remaining byte. The first-operand location is considered exhausted when it does not contain at least four remaining bytes.

When the second-operand location is exhausted, condition code 0 is set. When the first-operand location is exhausted, condition code 1 is set, except that condition code 0 is set if the second-operand location also is exhausted. When a CPU-determined number of characters have been processed, condition code 3 is set.

When the contents of the first byte of the next UTF-8 character are in the range 80 to BF hex or F8 to FF hex, the character is invalid, and condition code 2 is set.

When the conditions for setting condition codes 1 and 2 are both met, condition code 2 is set.

When the operation is completed, the contents of general register R₂ + 1 are decremented by the number of bytes converted, and the contents of general register R₂ are incremented by the same number. Also, the contents of general register R₁ + 1 are decremented by the number of bytes placed at the first-operand location, and the contents of general register R₁ are incremented by the same number. When general registers R₁ and R₂ are updated in the 24-bit or 31-bit addressing mode, bits 32-39 of them, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R₁, R₁ + 1, R₂, and R₂ + 1, always remain unchanged.

When condition code 2 is set, general register R₂ contains the address of the invalid UTF-8 character.

When condition code 3 is set, the registers have been updated so that the instruction, when reexecuted, resumes at the next byte locations to be processed.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the R₁ register is the same register as the R₂ register, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portions of the operands to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

When the length of an operand is zero, no access exceptions are recognized for that operand.

Access exceptions are not recognized for an
operand if the R field associated with that operand is odd.

**Resulting Condition Code:**

0  Entire second operand processed  
1  End of first operand reached  
2  Invalid UTF-8 character  
3  CPU-determined number of characters processed

**Program Exceptions:**

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 3 is not installed)
- Specification

**Programming Notes:**

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the conversion. The program need not determine the number of first-operand or second-operand bytes that were processed.
2. Bits 0 and 1 of the continuation bytes of multiple-byte UTF-8 characters are not checked in order to improve the performance of the conversion. Therefore, invalid continuation bytes are not detected.
3. The storage-operand references of CONVERT UTF-8 TO UTF-32 may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98)

**COPY ACCESS**

**DIVIDE**

DR  \(R_1, R_2\)  [RR]

<table>
<thead>
<tr>
<th>'1D'</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

D  \(R_1, D_2(X_2, B_2)\)  [RX]

<table>
<thead>
<tr>
<th>'5D'</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The 64-bit first operand (the dividend) is divided by the 32-bit second operand (the divisor), and the 32-bit remainder and quotient are placed at the first-operand location.

The \(R_1\) field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The dividend is treated as a 64-bit signed binary integer. The leftmost 32 bits of the dividend are in bit positions 32-63 of general register \(R_1\), and the rightmost 32 bits are in bit positions 32-63 of general register \(R_1 + 1\).

The divisor, remainder, and quotient are treated as 32-bit signed binary integers. For DIVIDE (DR), the divisor is in bit positions 32-63 of general register \(R_2\). The remainder is placed in bit positions 32-63 of general register \(R_1\), and the quotient is placed in bit positions 32-63 of general register \(R_1 + 1\). Bits 0-31 of the registers remain unchanged.

The sign of the quotient is determined by the rules of algebra, and the remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed by a 32-bit signed binary integer, a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

**Condition Code:** The code remains unchanged.
Program Exceptions:
- Access (fetch, operand 2 of D only)
- Fixed-point divide
- Specification

DIVIDE LOGICAL

DLR R₁,R₂ [RRE]

<table>
<thead>
<tr>
<th>'B997'</th>
<th>//////////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

DLGR R₁,R₂ [RRE]

<table>
<thead>
<tr>
<th>'B987'</th>
<th>//////////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

DL R₁,D₂(X₂,B₂) [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'97'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

DLG R₁,D₂(X₂,B₂) [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'87'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The 64-bit or 128-bit first operand (the dividend) is divided by the 32-bit or 64-bit second operand (the divisor), and the 32-bit or 64-bit remainder and quotient are placed at the first-operand location.

The R₁ field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

For DIVIDE LOGICAL (DLR, DL), the dividend is treated as a 64-bit unsigned binary integer. The leftmost 32 bits of the dividend are in bit positions 32-63 of general register R₁, and the rightmost 32 bits are in bit positions 32-63 of general register R₁ + 1.

32-63 of general register R₂. The remainder is placed in bit positions 32-63 of general register R₁, and the quotient is placed in bit positions 32-63 of general register R₁ + 1. Bits 0-31 of the registers remain unchanged.

For DIVIDE LOGICAL (DLGR, DLG), the dividend is treated as a 128-bit unsigned binary integer. The leftmost 64 bits of the dividend are in general register R₁, and the rightmost 64 bits are in general register R₁ + 1. The divisor, remainder, and quotient are treated as 64-bit unsigned binary integers. The remainder is placed in general register R₁, and the quotient is placed in general register R₁ + 1.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed as a 32-bit unsigned binary integer for DIVIDE LOGICAL (DLR, DL), or a 64-bit unsigned binary integer for DIVIDE LOGICAL (DLGR, DLG), a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2 of DL or DLG only)
- Fixed-point divide
- Specification

DIVIDE SINGLE

DSGR R₁,R₂ [RRE]

<table>
<thead>
<tr>
<th>'B90D'</th>
<th>//////////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

DSGFR R₁,R₂ [RRE]

<table>
<thead>
<tr>
<th>'B91D'</th>
<th>//////////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

DSG R₁,D₂(X₂,B₂) [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'0D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>
The 64-bit contents of general register \( R_{1} + 1 \) (the dividend) are divided by the 64-bit or 32-bit second operand (the divisor), the 64-bit remainder is placed in general register \( R_{1} \), and the 64-bit quotient is placed in general register \( R_{1} + 1 \).

The \( R_{1} \) field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The dividend, quotient, and remainder are treated as 64-bit signed binary integers. For DIVIDE SINGLE (DSGR, DSG), the divisor is treated as a 64-bit signed binary integer. For DIVIDE SINGLE (DSGFR, DSGF), the divisor is treated as a 32-bit signed binary integer. For DSGFR, the divisor is in bit positions 32-63 of general register \( R_{2} \).

The sign of the quotient is determined by the rules of algebra, and the remainder has the same sign as the dividend, except that a zero quotient or a zero remainder is always positive.

When the divisor is zero, or when the magnitudes of the dividend and divisor are such that the quotient cannot be expressed by a 64-bit signed binary integer, a fixed-point-divide exception is recognized. This includes the case of division of zero by zero.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of DSG and DSGF only)
- Fixed-point divide
- Specification

The EXCLUSIVE OR of the first and second operands is placed at the first-operand location.
The connective EXCLUSIVE OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the bits in the corresponding bit positions in the two operands are unlike; otherwise, the result bit is set to zero.

For EXCLUSIVE OR (XC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

For EXCLUSIVE OR (XI, XIX), the first operand is one byte in length, and only one byte is stored.

For EXCLUSIVE OR (XR, X, XY), the operands are 32 bits, and for EXCLUSIVE OR (XGR, XG), they are 64 bits.

The displacements for X, XI, and both operands of XC are treated as 12-bit unsigned binary integers. The displacement for XY, XIX, and XG is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

0  Result zero  
1  Result not zero  
2  --  
3  --

**Program Exceptions:**

- Access (fetch, operand 2, X, XY, and XC; fetch and store, operand 1, XI, XIX, and XC)
- Operation (XY and XIIX, if the long-displacement facility is not installed)

**Programming Notes:**

1. An example of the use of the EXCLUSIVE OR instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. EXCLUSIVE OR may be used to invert a bit, an operation particularly useful in testing and setting programmed binary switches.

3. A field EXCLUSIVE-ORed with itself becomes all zeros.

4. For EXCLUSIVE OR (XR or XGR), the sequence A EXCLUSIVE-OR B, B EXCLUSIVE-OR A, A EXCLUSIVE-OR B results in the exchange of the contents of A and B without the use of an additional general register.

5. Accesses to the first operand of EXCLUSIVE OR (XI) and EXCLUSIVE OR (XC) consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, EXCLUSIVE OR cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in “Multiprogramming and Multiprocessing Examples” in Appendix A, “Number Representation and Instruction-Use Examples.”

**EXECUTE**

```
EX  R1,D2(X2,B2)  [RX]
```

The single instruction at the second-operand address is modified by the contents of general register R1, and the resulting instruction, called the target instruction, is executed.

When the R1 field is not zero, bits 8-15 of the instruction designated by the second-operand address are ORed with bits 56-63 of general register R1. The ORing does not change either the contents of general register R1 or the instruction in storage, and it is effective only for the interpretation of the instruction to be executed. When the R1 field is zero, no ORing takes place.

The target instruction may be two, four, or six bytes in length. The execution and exception handling of the target instruction are exactly as if the target instruction were obtained in normal sequential operation, except for the instruction address and the instruction-length code.

The instruction address in the current PSW is increased by the length of EXECUTE. This updated address and the instruction-length code of EXECUTE are used, for example, as part of the link information when the target instruction is BRANCH AND LINK. When the target instruction
is a successful branching instruction, the instruction address in the current PSW is replaced by the branch address specified by the target instruction.

When the target instruction is in turn EXECUTE, an execute exception is recognized.

The effective address of EXECUTE must be even; otherwise, a specification exception is recognized. When the target instruction is two or three halfwords in length but can be executed without fetching its second or third halfword, it is unpredictable whether access exceptions are recognized for the unused halfwords. Access exceptions are not recognized for the second-operand address when the address is odd.

The second-operand address of EXECUTE is an instruction address rather than a logical address; thus, the target instruction is fetched from the primary address space when in the primary-space, secondary-space, or access-register mode.

**Condition Code:** The code may be set by the target instruction.

**Program Exceptions:**
- Access (fetch, target instruction)
- Execute
- Specification

**Programming Notes:**
1. An example of the use of the EXECUTE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. The ORing of eight bits from the general register with the designated instruction permits the indirect specification of the length, index, mask, immediate-data, register, or extended-op-code field.
3. The fetching of the target instruction is considered to be an instruction fetch for purposes of program-event recording and for purposes of reporting access exceptions.
4. An access or specification exception may be caused by EXECUTE or by the target instruction.
5. When an interruptible instruction is made the target of EXECUTE, the program normally should not designate any register updated by the interruptible instruction as the $R_1$, $X_2$, or $B_2$ register for EXECUTE. Otherwise, on resumption of execution after an interruption, or if the instruction is refetched without an interruption, the updated values of these registers will be used in the execution of EXECUTE. Similarly, the program should normally not let the destination field in storage of an interruptible instruction include the location of EXECUTE, since the new contents of the location may be interpreted when resuming execution.

**EXTRACT ACCESS**

**EXTRACT PSW**
**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**INSERT CHARACTER**

IC  \( R_1, D_2(X_2, B_2) \) [RX]

\[
\begin{array}{cccccc}
\text{'43'} & R & x_2 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 31
\end{array}
\]

ICY  \( R_1, D_2(X_2, B_2) \) [RXY]

\[
\begin{array}{cccccccc}
\text{'E3'} & R & x_2 & B_2 & DL_2 & DH_2 & '73' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47
\end{array}
\]

The byte at the second-operand location is inserted into bit positions 56-63 of general register \( R_1 \). The remaining bits in the register remain unchanged.

The displacement for IC is treated as a 12-bit unsigned binary integer. The displacement for ICY is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation (ICY, if the long-displacement facility is not installed)

**INSERT CHARACTERS UNDER MASK**

ICM  \( R_1, M_3, D_2(B_2) \) [RS]

\[
\begin{array}{cccccc}
\text{'BF'} & R & M_3 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 31
\end{array}
\]

ICMY  \( R_1, M_3, D_2(B_2) \) [RSY]

\[
\begin{array}{cccccccc}
\text{'EB'} & R & M_3 & B_2 & DL_2 & DH_2 & '81' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47
\end{array}
\]

Bytes from contiguous locations beginning at the second-operand address are inserted into general register \( R_1 \) under control of a mask.

The contents of the \( M_3 \) field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register \( R_1 \). For INSERT CHARACTERS UNDER MASK (ICM, ICMY), the four bytes to which the mask bits correspond are in the low-order half, bit positions 32-63 of general register \( R_1 \). For INSERT CHARACTERS UNDER MASK (ICMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The byte positions corresponding to ones in the mask are filled, left to right, with bytes from successive storage locations beginning at the second-operand address. When the mask is not zero, the length of the second operand is equal to the number of ones in the mask. The bytes in the general register corresponding to zeros in the mask remain unchanged. For ICM and ICMY, bits 0-31 of the register remain unchanged, and, for ICMH, bits 32-63 remain unchanged.

The resulting condition code is based on the mask and on the value of the bits inserted. When the mask is zero or when all inserted bits are zeros, the condition code is set to 0. When the inserted bits are not all zeros, the code is set according to the leftmost bit of the storage operand: if this bit is one, the code is set to 1; if this bit is zero, the code is set to 2.

When the mask is not zero, exceptions associated with storage-operand access are recognized only for the number of bytes specified by the mask. When the mask is zero, access exceptions are recognized for one byte at the second-operand address.

The displacement for ICM is treated as a 12-bit unsigned binary integer. The displacement for ICMY and ICMH is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

0  All inserted bits zeros, or mask bits all zeros
1  Leftmost inserted bit one
2. Leftmost inserted bit zero, and not all inserted bits zeros

3. --

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation (ICMY, if the long-displacement facility is not installed)

**Programming Notes:**
1. Examples of the use of the INSER T CHARACTERS UNDER MASK instruction are given in Appendix A, "Number Representation and Instruction-Use Examples."
2. The condition code for INSERT CHARACTERS UNDER MASK (ICM, ICMY only) is defined such that, when the mask is 1111, the instruction causes the same condition code to be set as for LOAD AND TEST (LTR only). Thus, the instruction may be used as a storage-to-register load-and-test operation.
3. INSERT CHARACTERS UNDER MASK (ICM, ICMY) with a mask of 1111 or 0001 performs a function similar to that of a LOAD (L) or INSERT CHARACTER (IC) instruction, respectively, with the exception of the condition-code setting. However, the performance of INSERT CHARACTERS UNDER MASK may be slower.

**INSERT IMMEDIATE**

IIHH R_{1,2} [RI]

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R_{1}</th>
<th>'0'</th>
<th>I_{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

IIHL R_{1,2} [RI]

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R_{1}</th>
<th>'1'</th>
<th>I_{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

IILH R_{1,2} [RI]

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R_{1}</th>
<th>'2'</th>
<th>I_{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

IILL R_{1,2} [RI]

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R_{1}</th>
<th>'3'</th>
<th>I_{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

The second operand is placed in bit positions of the first operand. The remainder of the first operand remains unchanged.

For each instruction, the bit positions of the first operand that are loaded with the second operand are as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIHH</td>
<td>0-15</td>
</tr>
<tr>
<td>IIHL</td>
<td>16-31</td>
</tr>
<tr>
<td>IILH</td>
<td>32-47</td>
</tr>
<tr>
<td>IILL</td>
<td>48-63</td>
</tr>
</tbody>
</table>

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**INSERT PROGRAM MASK**

IPM R_{1} [RRE]

<table>
<thead>
<tr>
<th>'B222'</th>
<th>///////</th>
<th>R_{1}</th>
<th>///</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The condition code and program mask from the current PSW are inserted into bit positions 34 and 35 and 36-39, respectively, of general register R_{1}. Bits 32 and 33 of the register are set to zeros; bits 0-31 and 40-63 are left unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.
LOAD

LR R₁, R₂ [RR]

<table>
<thead>
<tr>
<th>'18'</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

LGR R₁, R₂ [RRE]

<table>
<thead>
<tr>
<th>'B904'</th>
<th></th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>31</td>
</tr>
</tbody>
</table>

LGFR R₁, R₂ [RRE]

<table>
<thead>
<tr>
<th>'B914'</th>
<th></th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>31</td>
</tr>
</tbody>
</table>

L R₁, D₂(X₂, B₂) [RX]

<table>
<thead>
<tr>
<th>'58'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

LY R₁, D₂(X₂, B₂) [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'58'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

LG R₁, D₂(X₂, B₂) [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'04'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

LGF R₁, D₂(X₂, B₂) [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'14'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The second operand is placed unchanged at the first-operand location, except that, for LOAD (LGR, LG), it is sign extended.

For LOAD (LR, L, LY), the operands are 32 bits, and, for LOAD (LGR, LG), the operands are 64 bits. For LOAD (LGFR, LGF), the second operand is treated as a 32-bit signed binary integer, and the first operand is treated as a 64-bit signed binary integer.

The displacement for L is treated as a 12-bit unsigned binary integer. The displacement for LY, LG, and LGF is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of L, LY, LG, and LGF only)
- Operation (LY, if the long-displacement facility is not installed)

**Programming Note:** An example of the use of the LOAD instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

LOAD ACCESS MULTIPLE

LAM R₁, R₃, D₂(B₂) [RS]

<table>
<thead>
<tr>
<th>'9A'</th>
<th>R₁</th>
<th>R₃</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

LAMY R₁, R₃, D₂(B₂) [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R₁</th>
<th>R₃</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'9A'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The set of access registers starting with access register R₁ and ending with access register R₃ is loaded from the locations designated by the second-operand address.

The storage area from which the contents of the access registers are obtained starts at the location designated by the second-operand address and continues through as many storage words as the number of access registers specified. The access registers are loaded in ascending order of their register numbers, starting with access register R₁ and continuing up to and including access register R₃, with access register 0 following access register 15.

The displacement for LAM is treated as a 12-bit unsigned binary integer. The displacement for LAMY is treated as a 20-bit signed binary integer.
The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation (LAMY, if the long-displacement facility is not installed)
- Specification

### LOAD ADDRESS

**LOAD ADDRESS**

\[
\text{LA } R_1, D_2(X_2, B_2) \quad [RX]
\]

<table>
<thead>
<tr>
<th>'41'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'71'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The address specified by the \( X_2 \), \( B_2 \), and \( D_2 \) fields is placed in general register \( R_1 \). The address computation follows the rules for address arithmetic.

In the 24-bit addressing mode, the address is placed in bit positions 40-63, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The displacement for LA is treated as a 12-bit unsigned binary integer. The displacement for LAY is treated as a 20-bit signed binary integer.

No storage references for operands take place, and the address is not inspected for access exceptions.

**Condition Code:** The code remains unchanged.

### LOAD ADDRESS EXTENDED

**LOAD ADDRESS EXTENDED**

\[
\text{LAE } R_1, D_2(X_2, B_2) \quad [RX]
\]

<table>
<thead>
<tr>
<th>'51'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The address specified by the \( X_2 \), \( B_2 \), and \( D_2 \) fields is placed in general register \( R_1 \). Access register \( R_1 \) is loaded with a value that depends on the current value of the address-space-control bits, bits 16 and 17 of the PSW. If the address-space-control bits are 01 binary, the value placed in the access register also depends on whether the \( B_2 \) field is zero or nonzero.

The address computation follows the rules for address arithmetic. In the 24-bit addressing mode, the address is placed in bit positions 40-63 of general register \( R_1 \), bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The value placed in access register \( R_1 \) is as shown in the following table:

**Program Exceptions:**
- Operation (LAY if the long-displacement facility is not installed)

**Programming Notes:**
1. An example of the use of the LOAD ADDRESS instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. LOAD ADDRESS may be used to increment the rightmost bits of a general register, other than register 0, by the contents of the \( D_2 \) field of the instruction. LOAD ADDRESS (LAY) may also be used to decrement the rightmost bits of a register, other than register 0. The register to be incremented should be designated by \( R_1 \) and by either \( X_2 \) (with \( B_2 \) set to zero) or \( B_2 \) (with \( X_2 \) set to zero). The instruction updates 24 bits in the 24-bit addressing mode, 31 bits in the 31-bit addressing mode, and 64 bits in the 64-bit addressing mode.
However, when PSW bits 16 and 17 are 01 binary and the B\textsubscript{2} base/supertwo field is nonzero, bit positions 0-6 of access register B\textsubscript{2} must contain all zeros; otherwise, the results in general register R\textsubscript{1} and access register R\textsubscript{2} are unpredictable.

No storage references for operands take place, and the address is not inspected for access exceptions.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**Programming Notes:**

1. When DAT is on, the different values of the address-space-control bits correspond to translation modes as follows:

<table>
<thead>
<tr>
<th>PSW Bits 16 and 17</th>
<th>Value Placed in Access Register R\textsubscript{1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00000000 hex (zeros in bit positions 0-31)</td>
</tr>
<tr>
<td>10</td>
<td>00000001 hex (zeros in bit positions 0-30 and one in bit position 31)</td>
</tr>
<tr>
<td>01</td>
<td>If B\textsubscript{2} field is zero: 00000000 hex (zeros in bit positions 0-31)</td>
</tr>
<tr>
<td></td>
<td>If B\textsubscript{2} field is nonzero: Contents of access register B\textsubscript{2}</td>
</tr>
<tr>
<td>11</td>
<td>00000002 hex (zeros in bit positions 0-29 and 31, and one in bit position 30)</td>
</tr>
</tbody>
</table>

2. In the access-register mode, the value 00000000 hex in an access register designates the primary address space, and the value 00000001 hex designates the secondary address space. The value 00000002 hex designates the home address space if the control program assigns entry 2 of the dispatchable-unit access list as designating the home address space and places a zero access-list-entry sequence number (ALESN) in that entry.

### LOAD ADDRESS RELATIVE LONG

**LARL** \( R_1, I_2 \) \( [RIL] \)

<table>
<thead>
<tr>
<th>'C0'</th>
<th>R\textsubscript{1}</th>
<th>'0'</th>
<th>I\textsubscript{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

The address specified by the I\textsubscript{2} field is placed in general register R\textsubscript{1}. The address computation follows the rules for the branch address of BRANCH RELATIVE ON CONDITION LONG and BRANCH RELATIVE AND SAVE LONG.

In the 24-bit addressing mode, the address is placed in bit positions 40-63, bits 32-39 are set to zeros, and bits 0-31 remain unchanged. In the 31-bit addressing mode, the address is placed in bit positions 33-63, bit 32 is set to zero, and bits 0-31 remain unchanged. In the 64-bit addressing mode, the address is placed in bit positions 0-63.

The contents of the I\textsubscript{2} field are a signed binary integer specifying the number of halfwords that is added to the address of the instruction to generate the computed address.

No storage references for operands take place, and the address is not inspected for access exceptions.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**Programming Notes:**

1. Only even addresses (halfword addresses) can be generated. If an odd address is desired, LOAD ADDRESS can be used to add one to an address formed by LOAD ADDRESS RELATIVE LONG.

2. When LOAD ADDRESS RELATIVE LONG is the target of EXECUTE, the address produced is relative to the location of the LOAD ADDRESS RELATIVE LONG instruction, not of the EXECUTE instruction. This is consistent with the operation of the relative-branch instructions.
LOAD AND TEST

LTR  R₁, R₂  [RR]

\[
\begin{array}{c|cccc}
12 & R₁ & R₂ \\
\end{array}
\]

LTGR  R₁, R₂  [RRE]

\[
\begin{array}{c|cccc}
'B902' & / / / / / / & R₁ & R₂ \\
\end{array}
\]

LTGFR  R₁, R₂  [RRE]

\[
\begin{array}{c|cccc}
'B912' & / / / / / / & R₁ & R₂ \\
\end{array}
\]

The second operand is placed unchanged at the first-operand location, except that, for LOAD AND TEST (LTGFR), it is sign extended. The sign and magnitude of the second operand, treated as a signed binary integer, are indicated in the condition code.

For LOAD AND TEST (LTR), the operands are 32 bits, and, for LOAD AND TEST (LTGR), the operands are 64 bits. For LOAD AND TEST (LTGFR), the second operand is 32 bits, and the first operand is treated as a 64-bit signed binary integer.

**Resulting Condition Code:**

- 0 Result zero
- 1 Result less than zero
- 2 Result greater than zero
- 3 --

**Program Exceptions:** None.

**Programming Note:** For LOAD AND TEST (LTR and LTGR) when the R₁ and R₂ fields designate the same register, the operation is equivalent to a test without data movement.

LOAD BYTE

LB  R₁, D₂ (X₂, B₂)  [RXY]

\[
\begin{array}{c|cccccc}
'E3' & R₁ & X₂ & B₂ & DL₂ & DH₂ & '76' \\
\end{array}
\]

LGB  R₁, D₂ (X₂, B₂)  [RXY]

\[
\begin{array}{c|cccccc}
'E3' & R₁ & X₂ & B₂ & DL₂ & DH₂ & '77' \\
\end{array}
\]

The second operand is sign extended and placed at the first-operand location. The second operand is one byte in length and is treated as an eight-bit signed binary integer. For LOAD BYTE (LB), the first operand is treated as a 32-bit signed binary integer. For LOAD BYTE (LGB), the first operand is treated as a 64-bit signed binary integer.

The displacement is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2)
- Operation (if the long-displacement facility is not installed)

LOAD COMPLEMENT

LCR  R₁, R₂  [RR]

\[
\begin{array}{c|cccc}
13 & R₁ & R₂ \\
\end{array}
\]

LCGR  R₁, R₂  [RRE]

\[
\begin{array}{c|cccc}
'B903' & / / / / / / & R₁ & R₂ \\
\end{array}
\]

LCGFR  R₁, R₂  [RRE]

\[
\begin{array}{c|cccc}
'B913' & / / / / / / & R₁ & R₂ \\
\end{array}
\]

7-126  z/Architecture Principles of Operation
The two's complement of the second operand is placed at the first-operand location. For LOAD COMPLEMENT (LCR), the second operand and result are treated as 32-bit signed binary integers. For LOAD COMPLEMENT (LCGR), they are treated as 64-bit signed binary integers. For LOAD COMPLEMENT (LCGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

**Resulting Condition Code:**

0  Result zero; no overflow
1  Result less than zero; no overflow
2  Result greater than zero; no overflow
3  Overflow

**Program Exceptions:**

- Fixed-point overflow

**Programming Note:** The operation complements all numbers. Zero remains unchanged. For LCR or LCGR, the maximum negative 32-bit number or 64-bit number, respectively, remains unchanged, and an overflow condition occurs when the number is complemented. LCGFR complements the maximum negative 32-bit number without recognizing overflow.

**LOAD HALFWORD**

LH  \( R_1, D_2(X_2, B_2) \)  [RX]

<table>
<thead>
<tr>
<th>'48'</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

LHY \( R_1, D_2(X_2, B_2) \)  [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>DL2</th>
<th>DH2</th>
<th>'78'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The second operand is sign extended and placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. For LOAD HALFWORD (LH, LHY) and LOAD HALFWORD IMMEDIATE (LHI), the first operand is treated as a 32-bit signed binary integer. For LOAD HALFWORD (LGH) and LOAD HALFWORD IMMEDIATE (LGHI), the first operand is treated as a 64-bit signed binary integer.

The displacement for LH is treated as a 12-bit unsigned binary integer. The displacement for LHY and LGH is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2 of LH, LHY, and LGH)

**Program Exceptions:**

- Access (fetch, operand 2 of LH and LHY)
- Operation (LHY, if the long-displacement facility is not installed)

**Programming Note:** An example of the use of the LOAD HALFWORD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
**LOAD LOGICAL**

**LOAD LOGICAL**

```
LLGFR R1, R2 [RRE]
```

<table>
<thead>
<tr>
<th>'B916'</th>
<th>///////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

```
LLGF R1, D2(Xz, Bz) [RXY]
```

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R1</th>
<th>Xz</th>
<th>Bz</th>
<th>DLz</th>
<th>DHz</th>
<th>'16'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The four-byte second operand is placed in bit positions 32-63 of general register R1, and zeros are placed in bit positions 0-31 of general register R1.

For LOAD LOGICAL (LLGFR), the second operand is in bit positions 32-63 of general register R2.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of LLGF only)

**LOAD LOGICAL CHARACTER**

```
LLGC R1, D2(Xz, Bz) [RXY]
```

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R1</th>
<th>Xz</th>
<th>Bz</th>
<th>DLz</th>
<th>DHz</th>
<th>'90'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The one-byte second operand is placed in bit positions 56-63 of general register R1, and zeros are placed in bit positions 0-55 of general register R1.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)

**LOAD LOGICAL HALFWORD**

```
LLGH R1, D2(Xz, Bz) [RXY]
```

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R1</th>
<th>Xz</th>
<th>Bz</th>
<th>DLz</th>
<th>DHz</th>
<th>'91'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The two-byte second operand is placed in bit positions 48-63 of general register R1, and zeros are placed in bit positions 0-47 of general register R1.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)

**LOAD LOGICAL IMMEDIATE**

```
LLIH R1, I2 [RI]
```

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R1</th>
<th>'C'</th>
<th>I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

```
LLII R1, I2 [RI]
```

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R1</th>
<th>'D'</th>
<th>I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

```
LLIL R1, I2 [RI]
```

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R1</th>
<th>'E'</th>
<th>I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

```
LLIL R1, I2 [RI]
```

<table>
<thead>
<tr>
<th>'A5'</th>
<th>R1</th>
<th>'F'</th>
<th>I2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

The second operand is placed in bit positions of the first operand. The remainder of the first operand is set to zeros.

For each instruction, the bit positions of the first operand that are loaded with the second operand are as follows:
### Condition Code:
The code remains unchanged.

### Program Exceptions:
None.

#### LOAD LOGICAL THIRTY ONE BITS

**LLGTR**  \( R_1, R_2 \)  \( [RRE] \)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bit Positions Loaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLIHH</td>
<td>0-15</td>
</tr>
<tr>
<td>LLIHL</td>
<td>16-31</td>
</tr>
<tr>
<td>LLIILH</td>
<td>32-47</td>
</tr>
<tr>
<td>LLIILL</td>
<td>48-63</td>
</tr>
</tbody>
</table>

For **LLGTR**, bits 33-63 of general register \( R_2 \), with 33 zeros appended on the left, are placed in general register \( R_1 \). For **LLGT**, bits 1-31 of the four bytes at the second-operand location, with 33 zeros appended on the left, are placed in general register \( R_1 \).

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of LLGT only)

#### LOAD MULTIPLE

**LM**  \( R_1, R_3, D_2 (B_2) \)  \( [RS] \)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bit Positions Loaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMY</td>
<td>'EB' ( R_1, R_3, D_2 (B_2) )  ( [RSY] )</td>
</tr>
<tr>
<td>LMG</td>
<td>'EB' ( R_1, R_3, D_2 (B_2) )  ( [RSY] )</td>
</tr>
</tbody>
</table>

Bit positions of the set of general registers starting with general register \( R_1 \) and ending with general register \( R_3 \) are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed.

For **LOAD MULTIPLE** (LM, LMY), bit positions 32-63 of the general registers are loaded from successive four-byte fields beginning at the second-operand address, and bits 0-31 of the registers remain unchanged. For **LOAD MULTIPLE** (LMG), bit positions 0-63 of the general registers are loaded from successive eight-byte fields beginning at the second-operand address.

The general registers are loaded in the ascending order of their register numbers, starting with general register \( R_1 \) and continuing up to and including general register \( R_3 \), with general register 0 following general register 15.

The displacement for LM is treated as a 12-bit unsigned binary integer. The displacement for LMY and LMG is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Operation (LMY, if the long-displacement facility is not installed)

**Programming Note:** All combinations of register numbers specified by \( R_1 \) and \( R_3 \) are valid. When the register numbers are equal, only four bytes, for LM or LMY or eight bytes, for LMG, are transmitted. When the number specified by \( R_3 \) is less than the number specified by \( R_1 \), the register numbers wrap around from 15 to 0.
Bit positions 0-31 of the set of general registers starting with general register $R_1$ and ending with general register $R_3$ are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed. Bit positions 32-63 of the same registers are similarly loaded from storage beginning at the location designated by the fourth-operand address.

The general registers are loaded in the ascending order of their register numbers, starting with general register $R_1$ and continuing up to and including general register $R_3$, with general register 0 following general register 15.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operands 2 and 4)

**Programming Notes:**
1. All combinations of register numbers specified by $R_1$ and $R_3$ are valid. When the register numbers are equal, only eight bytes are transmitted. When the number specified by $R_3$ is less than the number specified by $R_1$, the register numbers wrap around from 15 to 0.
2. The second-operand and fourth-operand addresses are computed before the contents of any register are changed.
3. The combination of a LOAD MULTIPLE instruction and a LOAD MULTIPLE HIGH instruction provides equal or better performance than a LOAD MULTIPLE DISJOINT instruction for the same register range. LOAD MULTIPLE DISJOINT is for use when the second or fourth operand must be addressed by means of one of the registers loaded.
The two's complement of the absolute value of the second operand is placed at the first-operand location. For LOAD NEGATIVE (LNR), the second operand and result are treated as 32-bit signed binary integers, and, for LOAD NEGATIVE (LNGR), they are treated as 64-bit signed binary integers. For LOAD NEGATIVE (LNGFR), the second operand is treated as a 32-bit signed binary integer, and the result is treated as a 64-bit signed binary integer.

**Resulting Condition Code:**

- 0 Result zero
- 1 Result less than zero
- 2 --
- 3 --

**Program Exceptions:** None.

**Programming Note:** The operation complements positive numbers; negative numbers remain unchanged. The number zero remains unchanged.

### LOAD PAIR FROM QUADWORD

The quadword second operand is loaded into the first-operand location. The second operand appears to be fetched with quadword concurrency as observed by other CPUs. The left doubleword of the quadword is loaded into general register \( R_1 \), and the right doubleword is loaded into general register \( R_1 + 1 \).

The \( R_1 \) field designates an even-odd pair of general registers and must designate an even-numbered register. The second operand must be designated on a quadword boundary. Otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

### LOAD POSITIVE

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

**Program Exceptions:**

- Access (fetch, operand 2)
- Specification

**Programming Notes:**

1. The LOAD MULTIPLE (LM or LMG) instruction does not necessarily provide quadword-concurrent access.
2. The performance of LOAD PAIR FROM QUADWORD on some models may be significantly slower than that of LOAD MULTIPLE (LMG). Unless quadword consistency is required, LMG should be used instead of LPQ.
**Resulting Condition Code:**

0  Result zero; no overflow  
1  --  
2  Result greater than zero; no overflow  
3  Overflow  

**Program Exceptions:**

- Fixed-point overflow  

**Programming Note:** The operation complements negative numbers; positive numbers and zero remain unchanged. For LPR or LPGR, an overflow condition occurs when the maximum negative 32-bit number or 64-bit number, respectively, is complemented; the number remains unchanged. LPGFR complements the maximum negative 32-bit number without recognizing overflow.

**LOAD REVERSED**

`LRVR R1,R2 [RRE]`

```
'B91F' //////    R1 R2
0  16  24  28  31
```

`LRVGR R1,R2 [RRE]`

```
'B90F' //////    R1 R2
0  16  24  28  31
```

`LRVH R1,Dz(Xz,Bz) [RXY]`

```
'E3' R1 Xz Bz DLz DHz '1F'
0  8 12 16 20 32 40 47
```

`LRV R1,Dz(Xz,Bz) [RXY]`

```
'E3' R1 Xz Bz DLz DHz '1E'
0  8 12 16 20 32 40 47
```

`LRVG R1,Dz(Xz,Bz) [RXY]`

```
'E3' R1 Xz Bz DLz DHz '/zerodotF'
0  8 12 16 20 32 40 47
```

The second operand is placed at the first-operand location with the left-to-right sequence of the bytes reversed.

For LOAD REVERSED (LRVH), the second operand is two bytes, the result is placed in bit positions 48-63 of general register R, and bits 0-47 of the register remain unchanged.

For LOAD REVERSED (LRVR, LRV), the second operand is four bytes, the result is placed in bit positions 32-63 of general register R, and bits 0-31 of the register remain unchanged. For LOAD REVERSED (LRVR), the second operand is in bit positions 32-63 of general register R.

For LOAD REVERSED (LRVGR, LRVG), the second operand is eight bytes.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2 of LRVH, LRV, LRVG only)

**Programming Notes:**

1. The instruction can be used to convert two, four, or eight bytes from a “little-endian” format to a “big-endian” format, or vice versa. In the big-endian format, the bytes in a left-to-right sequence are in the order most significant to least significant. In the little-endian format, the bytes are in the order least significant to most significant. For example, the bytes ABCD in the big-endian format are DCBA in the little-endian format.

2. LOAD REVERSED (LRVR) can be used with a two-byte value already in a register as shown in the following example. In the example, the two bytes of interest are in bit positions 48-63 of the R1 register.

```
LRVR R1,R1
SRA R1,16
```

The LOAD REVERSED instruction places the two bytes of interest in bit positions 32-47 of the register, with the order of the bytes reversed. The SHIFT RIGHT SINGLE (SRA) instruction shifts the two bytes to bit positions 48-63 of the register and extends them on their left, in bit positions 32-47, with their sign bit. The instruction SHIFT RIGHT SINGLE LOGICAL (SRL) should be used, instead, if the two bytes of interest are unsigned.
3. The storage-operand references of LOAD REVERSED may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98.)

**MONITOR CALL**

MC D1(B1),I2 [SI]

<table>
<thead>
<tr>
<th>'AF'</th>
<th>I2</th>
<th>B1</th>
<th>D1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

A program interruption is caused if the appropriate monitor-mask bit in control register 8 is one.

The monitor-mask bits are in bit positions 48-63 of control register 8, which correspond to monitor classes 0-15, respectively.

Bit positions 12-15 in the I2 field contain a binary number specifying one of 16 monitoring classes. When the monitor-mask bit corresponding to the class specified by the I2 field is one, a monitor-event program interruption occurs. The contents of the I2 field are stored at location 149, with zeros stored at location 148. Bit 9 of the program-interruption code is set to one.

The first-operand address is not used to address data; instead, the address specified by the B1 and D1 fields forms the monitor code, which is placed in the doubleword at location 176. Address computation follows the rules of address arithmetic; in the 24-bit addressing mode, bits 0-39 are set to zeros; in the 31-bit addressing mode, bits 0-32 are set to zeros.

When the monitor-mask bit corresponding to the class specified by bits 12-15 of the instruction is zero, no interruption occurs, and the instruction is executed as a no-operation.

Bit positions 8-11 of the instruction must contain zeros; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Monitor event
- Specification

**Programming Notes:**

1. **MONITOR CALL** provides the capability for passing control to a monitoring program when selected points are reached in the monitored program. This is accomplished by implanting MONITOR CALL instructions at the desired points in the monitored program. This function may be useful in performing various measurement functions; specifically, tracing information can be generated indicating which programs were executed, counting information can be generated indicating how often particular programs were used, and timing information can be generated indicating the amount of time a particular program required for execution.

2. The monitor masks provide a means of disallowing all monitor-event program interruptions or allowing monitor-event program interruptions for all or selected classes.

3. The monitor code provides a means of associating descriptive information, in addition to the class number, with each MONITOR CALL. Without the use of a base register, up to 4,096 distinct monitor codes can be associated with a monitoring interruption. With the base register designated by a nonzero value in the B1 field, each monitoring interruption can be identified by a 24-bit, 31-bit, or 64-bit code, depending on the addressing mode.

**MOVE**

MVI D1(B1),I2 [SI]

<table>
<thead>
<tr>
<th>'92'</th>
<th>I2</th>
<th>B1</th>
<th>D1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

MVIY D1(B1),I2 [SIY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>I2</th>
<th>B1</th>
<th>DL1</th>
<th>DH1</th>
<th>'52'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

MVC D1(L,B1),D2(B2) [SS]

<table>
<thead>
<tr>
<th>'D2'</th>
<th>L</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

Chapter 7. General Instructions 7-133
The second operand is placed at the first-operand location.

For MOVE (MVC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand byte.

For MOVE (MVI, MVIY), the first operand is one byte in length, and only one byte is stored.

The displacements for MVI and both operands of MVC are treated as 12-bit unsigned binary integers. The displacement for MVIY is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of MVC; store, operand 1, MVI, MVIY, and MVC)
- Operation (MVIY, if the long-displacement facility is not installed)

**Programming Notes:**
1. Examples of the use of the MOVE instruction are given in Appendix A, "Number Representation and Instruction-Use Examples."
2. It is possible to propagate one byte through an entire field by having the first operand start one byte to the right of the second operand.

**MOVE INVERSE**

MVCIN D₁(L₂,B₁),D₂(B₂) [SS]

<table>
<thead>
<tr>
<th>'E8'</th>
<th>L</th>
<th>B₁</th>
<th>D₁</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The second operand is placed at the first-operand location with the left-to-right sequence of the bytes inverted.

The first-operand address designates the leftmost byte of the first operand. The second-operand address designates the rightmost byte of the second operand. Both operands have the same length.

The result is obtained as if the second operand were processed from right to left and the first operand from left to right. The second operand may wrap around from location 0 to location $2^{24} - 1$ in the 24-bit addressing mode, to location $2^{31} - 1$ in the 31-bit addressing mode, or to location $2^{64} - 1$ in the 64-bit addressing mode. The first operand may wrap around from location $2^{24} - 1$ to location 0 in the 24-bit addressing mode, from location $2^{31} - 1$ to location 0 in the 31-bit addressing mode, or from location $2^{64} - 1$ to location 0 in the 64-bit addressing mode.

When the operands overlap by more than one byte, the contents of the overlapped portion of the result field are unpredictable.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2; store, operand 1)

**Programming Notes:**
1. An example of the use of the MOVE INVERSE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. The contents of each byte moved remain unchanged.
3. MOVE INVERSE is the only SS-format instruction for which the second-operand address designates the rightmost, instead of the leftmost, byte of the second operand.
4. The storage-operand references for MOVE INVERSE may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98.)

**MOVE LONG**

MVCL R₁,R₂ [RR]

<table>
<thead>
<tr>
<th>'0E'</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

The second operand is placed at the first-operand location, provided overlapping of operand locations would not affect the final contents of the first-operand location. The remaining rightmost
byte positions, if any, of the first-operand location are filled with padding bytes.

The \( R_1 \) and \( R_2 \) fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers \( R_1 \) and \( R_2 \), respectively. The number of bytes in the first-operand and second-operand locations is specified by unsigned binary integers in bit positions 40-63 of general registers \( R_1 + 1 \) and \( R_2 + 1 \), respectively. Bit positions 32-39 of general register \( R_2 + 1 \) contain the padding byte. The contents of bit positions 0-39 of general register \( R_1 + 1 \) and of bit positions 0-31 of general register \( R_2 + 1 \) are ignored.

The handling of the addresses in general registers \( R_1 \) and \( R_2 \) is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers \( R_1 \) and \( R_2 \) constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The contents of the registers just described are shown in Figure 7-2 on page 7-136.

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, byte by byte. The operation is ended when the number of bytes specified by bits 40-63 of general register \( R_1 + 1 \) have been moved into the first-operand location. If the second operand is shorter than the first operand, the remaining rightmost bytes of the first-operand location are filled with the padding byte.

As part of the execution of the instruction, the values of the two length fields are compared for the setting of the condition code, and a check is made for destructive overlap of the operands. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it, assuming the inspection for overlap is performed by the use of logical operand addresses. When the operands overlap destructively, no movement takes place, and condition code 3 is set.

Operands do not overlap destructively, and movement is performed, if the leftmost byte of the first operand does not coincide with any of the second-operand bytes participating in the operation other than the leftmost byte of the second operand. When an operand wraps around from location \( 2^{24} - 1 \) (or \( 2^{31} - 1 \) or \( 2^{64} - 1 \)) to location 0, operand bytes in locations up to and including \( 2^{24} - 1 \) (or \( 2^{31} - 1 \) or \( 2^{64} - 1 \)) are considered to be to the left of bytes in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location \( 2^{24} - 1 \) to location 0; in the 31-bit addressing mode, wraparound is from location \( 2^{31} - 1 \) to location 0; in the 64-bit addressing mode, wraparound is from location \( 2^{64} - 1 \) to location 0.

In the access-register mode, the contents of access register \( R_1 \) and access register \( R_2 \) are compared. If the \( R_1 \) or \( R_2 \) field is zero, 32 zeros are used rather than the contents of access register 0. If all 32 bits of the compared values are equal, then the destructive overlap test is made. If all 32 bits of the compared values are not equal, destructive overlap is declared not to exist. If, for this case, the operands actually overlap in real storage, it is unpredictable whether the result reflects the overlap condition.

When the length specified by bits 40-63 of general register \( R_1 + 1 \) is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

The execution of the instruction is interruptible. When an interruption occurs, other than one that follows termination, the lengths in general registers \( R_1 + 1 \) and \( R_2 + 1 \) are decremented by the number of bytes moved, and the addresses in general registers \( R_1 \) and \( R_2 \) are incremented by the same number, so that the instruction, when reexecuted, resumes at the point of interruption. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers \( R_1 \) and \( R_2 \) are set to zeros, and the contents of bit positions 0-31 remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers

Chapter 7. General Instructions 7-135
R₁ + 1 and R₂ + 1 remain unchanged; and the condition code is unpredictable. If the operation is interrupted during padding, the length field in general register R₂ + 1 is 0, the address in general register R₂ is incremented by the original length in general register R₂ + 1, and general registers R₁ and R₁ + 1 reflect the extent of the padding operation.

When the first-operand location includes the location of the instruction or of EXECUTE, the instruction may be refetched from storage and reinterpreted even in the absence of an interruption during execution. The exact point in the execution at which such a refetch occurs is unpredictable.

Padding byte values of B0 hex and B8 hex may be used during the nonpadding part of the operation by some models, in certain cases, as an indication of whether the movement should be performed bypassing the cache or using the cache, respectively. Thus, a padding byte of B0 hex indicates no intention to reference the destination.
area after the move, and a padding byte of B8 hex indicates an intention to reference the destination area.

For the nonpadding part of the operation, accesses to the operands for MOVE LONG are single-access references. These accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs, unless the padding byte is B1 hex. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by channel programs, that portion of the first operand which is filled with the padding byte is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once.

At the completion of the operation, the length in general register R1 + 1 is decremented by the number of bytes stored at the first-operand location, and the address in general register R1 is incremented by the same amount. The length in general register R2 + 1 is decremented by the number of bytes moved out of the second-operand location, and the address in general register R2 is incremented by the same amount. In the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R1 and R2 are set to zeros, even when one or both of the original length values are zeros or when condition code 3 is set. The contents of bit positions 0-31 of the registers remain unchanged. In any addressing mode, the contents of bit positions 0-39 of general registers R1 + 1 and R2 + 1 remain unchanged.

When condition code 3 is set, no exceptions associated with operand access are recognized. When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the second operand is longer than the first operand, access exceptions are not recognized for the part of the second-operand field that is in excess of the first-operand field. For operands longer than 2K bytes, access exceptions are not recognized for locations more than 2K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Also, when the R1 field is odd, PER storage-alteration events are not recognized, and no change bits are set.

Resulting Condition Code:
0 Operand lengths equal; no destructive overlap
1 First-operand length low; no destructive overlap
2 First-operand length high; no destructive overlap
3 No movement performed because of destructive overlap

Program Exceptions:
• Access (fetch, operand 2; store, operand 1)
• Specification

Programming Notes:
1. An example of the use of the MOVE LONG instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. MOVE LONG may be used for clearing storage by setting the padding byte to zero and the second-operand length to zero. On most models, this is the fastest instruction for clearing storage areas in excess of 256 bytes. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see “Storage-Operand Consistency” on page 5-98
3. The program should avoid specification of a length for either operand which would result in an addressing exception. Addressing (and also protection) exceptions may result in termination of the entire operation, not just the current unit of operation. The termination may be such that the contents of all result fields are unpredictable; in the case of MOVE LONG, this includes the condition code and the two even-odd general-register pairs, as well as the first-operand location in main storage. The following are situations that have actually occurred on one or more models:
a. When a protection exception occurs on a 4K-byte block of a first operand which is several blocks in length, stores to the protected block are suppressed. However, the move continues into the subsequent blocks of the first operand, which are not protected. Similarly, an addressing exception on a block does not necessarily suppress processing of subsequent blocks which are available.

b. Some models may update the general registers only when an external, I/O, repressible machine-check, or restart interruption occurs, or when a program interruption occurs for which it is required to nullify or suppress a unit of operation. Thus, if, after a move into several blocks of the first operand, an addressing or protection exception occurs, the general registers may remain unchanged.

4. When the first-operand length is zero, the operation consists in setting the condition code and, in the 24-bit or 31-bit addressing mode, of setting the leftmost bits in bit positions 32-63 of general registers R_{basesuperone} and R_{basesupertwo} to zero.

5. When the contents of the R_{basesuperone} and R_{basesupertwo} fields are the same, the contents of the designated registers are incremented or decremented only by the number of bytes moved, not by twice the number of bytes moved. Condition code 0 is set.

6. The following is a detailed description of those cases in which movement takes place, that is, where destructive overlap does not exist.

In the access-register mode, the contents of the access registers used are called the effective space designations. When the effective space designations are not equal, destructive overlap is declared not to exist and movement occurs. When the effective space designations are the same or when not in the access-register mode, then the following cases apply.

Depending on whether the second operand wraps around from location 2^{24} - 1 (or 2^{31} - 1 or 2^{64} - 1, depending on the addressing mode) to location 0, movement takes place in the following cases:

a. When the second operand does not wrap around, movement is performed if the leftmost byte of the first operand coincides with or is to the left of the leftmost byte of the second operand, or if the leftmost byte of the first operand is to the right of the rightmost second-operand byte participating in the operation.

b. When the second operand wraps around, movement is performed if the leftmost byte of the first operand coincides with or is to the left of the leftmost byte of the second operand, and if the leftmost byte of the first operand is to the right of the rightmost second-operand byte participating in the operation.

The rightmost second-operand byte is determined by using the smaller of the first-operand and second-operand lengths.

When the second-operand length is one or zero, destructive overlap cannot exist.

7. Special precautions should be taken if MOVE LONG is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

8. Since the execution of MOVE LONG is interruptible, the instruction cannot be used for situations where the program must rely on uninterrupted execution of the instruction. Similarly, the program should normally not let the first operand of MOVE LONG include the location of the instruction or of EXECUTE because the new contents of the location may be interpreted for a resumption after an interruption, or the instruction may be refetched without an interruption.

9. Further programming notes concerning interruptible instructions are included in the chapter on interruptible instructions in Chapter 5, “Program Execution.”

10. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.
MOVE LONG EXTENDED

MVCLE R₁,R₃,D₂(B₂) [RS]

<table>
<thead>
<tr>
<th>'A8'</th>
<th>R₁</th>
<th>R₃</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All or part of the third operand is placed at the first-operand location. The remaining rightmost byte positions, if any, of the first-operand location are filled with padding bytes. The operation proceeds until the end of the first-operand location is reached or a CPU-determined number of bytes have been placed at the first-operand location, whichever occurs first. The result is indicated in the condition code.

The R₁ and R₃ fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and third operand is designated by the contents of general registers R₁ and R₃, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers R₁ + 1 and R₃ + 1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the entire contents of general registers R₁ + 1 and R₃ + 1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The handling of the addresses in general registers R₁ and R₃ is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R₁ and R₃ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The second-operand address is not used to address data; instead, the rightmost eight bits of the second-operand address, bits 56-63, are the padding byte. Bits 0-55 of the second-operand address are ignored.

The contents of the registers and address just described are shown in Figure 7-73 on page 7-140.

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, byte by byte. The operation is ended when the number of bytes specified in general register R₁ + 1 have been placed at the first-operand location or when a CPU-determined number of bytes have been placed, whichever occurs first. If the third operand is shorter than the first operand, the remaining rightmost bytes of the first-operand location are filled with the padding byte.

When the operation is completed because the end of the first operand has been reached, the condition code is set to 0 if the two operand lengths are equal, it is set to 1 if the first-operand length is less than the third-operand length, or it is set to 2 if the first-operand length is greater than the third-operand length. When the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, condition code 3 is set.

No test is made for destructive overlap, and the results in the first-operand location are unpredictable when destructive overlap exists. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it.

Operands do not overlap destructively if the leftmost byte of the first operand does not coincide with any of the third-operand bytes participating in the operation other than the leftmost byte of the third operand. When an operand wraps around from location $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) to location 0, operand bytes in locations up to and including $2^{24} - 1$ (or $2^{31} - 1$ or $2^{64} - 1$) are considered to be to the left of bytes in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location $2^{24} - 1$ to location 0; in the 31-bit addressing mode, wraparound is from location $2^{31} - 1$ to location 0; and, in the 64-bit addressing mode, wraparound is from location $2^{64} - 1$ to location 0.
When the length specified in general register $R_{1} + 1$ is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

---

**Figure 7-73. Register Contents and Second-Operand Address for MOVE LONG EXTENDED**
Padding byte values of B0 hex and B8 hex may be used during the nonpadding part of the operation by some models, in certain cases, as an indication of whether the movement should be performed bypassing the cache or using the cache, respectively. Thus, a padding byte of B0 hex indicates no intention to reference the destination area after the move, and a padding byte of B8 hex indicates an intention to reference the destination area.

For the nonpadding part of the operation, accesses to the operands for MOVE LONG are single-access references. These accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs, unless the padding byte is B1 hex. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by channel programs, that portion of the first operand which is filled with the padding byte is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once.

At the completion of the operation, the length in general register R1 + 1 is decremented by the number of bytes stored at the first-operand location, and the address in general register R1 is incremented by the same amount. The length in general register R3 + 1 is decremented by the number of bytes moved out of the third-operand location, and the address in general register R3 is incremented by the same amount.

If the operation is completed because a CPU-determined number of bytes have been moved without reaching the end of the first operand, the lengths in general registers R1 + 1 and R3 + 1 are decremented by the number of bytes moved, and the addresses in general registers R1 and R3 are incremented by the same number, so that the instruction, when reexecuted, resumes at the next byte to be moved. If the operation is completed during padding, the length field in general register R3 + 1 is zero, the address in general register R3 is incremented by the original length in general register R3 + 1, and general registers R1 and R1 + 1 reflect the extent of the padding operation.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R1, R1 + 1, R3, and R3 + 1, always remain unchanged.

The padding byte may be formed from D2(B2) multiple times during the execution of the instruction, and the registers designated by R1 and R3 may be updated multiple times. Therefore, if B2 equals R1, R1 + 1, R3, or R3 + 1 and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed. The maximum amount is approximately 4K bytes of either operand.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers R1 and R3 may be set to zeros or may remain unchanged from their original values, even when one or both of the original length values are zeros.

When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the third operand is longer than the first operand, access exceptions are not recognized for the part of the third-operand field that is in excess of the first-operand field. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the R field associated with that operand is odd. Also, when the R1 field is odd, PEI storage-alteration events are not recognized, and no change bits are set.

**Resulting Condition Code:**

0  All bytes moved, operand lengths equal
1  All bytes moved, first-operand length low
2  All bytes moved, first-operand length high
3  CPU-determined number of bytes moved without reaching end of first operand
Program Exceptions:
- Access (fetch, operand 3; store, operand 1)
- Specification

Programming Notes:
1. MOVE LONG EXTENDED is intended for use in place of MOVE LONG when the operand lengths are specified as 32-bit or 64-bit binary integers and a test for destructive overlap is not required. MOVE LONG EXTENDED sets condition code 3 in cases in which MOVE LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the movement. The program need not determine the number of bytes that were moved.

3. The function of not processing more than approximately 4K bytes of either operand is intended to permit software polling of a flag that may be set by a program on another CPU during long operations.

4. MOVE LONG EXTENDED may be used for clearing storage by setting the padding byte to zero and the third-operand length to zero. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see “Storage-Operand Consistency” on page 5-98.

5. When the contents of the R1 and R3 fields are the same, the contents of the designated registers are incremented or decremented only by the number of bytes moved, not by twice the number of bytes moved. The condition code is finally set to 0 after possible settings to 3.

6. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

MOVE LONG UNICODE

MVCLU R1,R3,D2(B2) [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R1</th>
<th>R3</th>
<th>Bz</th>
<th>DLz</th>
<th>DHz</th>
<th>'8E'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

All or part of the third operand is placed at the first-operand location. The remaining rightmost two-byte character positions, if any, of the first-operand location are filled with two-byte padding characters. The operation proceeds until the end of the first-operand location is reached or a CPU-determined number of characters have been placed at the first-operand location, whichever occurs first. The result is indicated in the condition code.

The R1 and R3 fields each designate an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost character of the first operand and third operand is designated by the contents of general registers R1 and R3, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 32-63 of general registers R1+1 and R3+1, respectively, and those contents are treated as 32-bit unsigned binary integers. In the 64-bit addressing mode, the number of bytes in the first-operand and third-operand locations is specified by the contents of bit positions 0-63 of general registers R1+1 and R3+1, respectively, and those contents are treated as 64-bit unsigned binary integers.

The contents of general registers R1+1 and R3+1 must specify an even number of bytes; otherwise, a specification exception is recognized.

The handling of the addresses in general registers R1 and R3 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R1 and R3 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In
the 64-bit addressing mode, the contents of bit positions 0-63 of the registers constitute the address.

The second-operand address is not used to address data; instead, the rightmost 16 bits of the address.

**Figure 7-74. Register Contents and Second-Operand Address for MOVE LONG UNICODE**
The contents of the registers and address just described are shown in Figure 7-74 on page 7-143.

The result is obtained as if the movement starts at the left end of both fields and proceeds to the right, character by character. The operation is ended when the number of characters specified by the contents of general register \( R_1 + 1 \) have been placed at the first-operand location or when a CPU-determined number of characters have been placed, whichever occurs first. If the third operand is shorter than the first operand, the remaining rightmost character positions of the first-operand location are filled with the two-byte padding character.

When the operation completed because the end of the first operand has been reached, the condition code 3 is set.

No test is made for destructive overlap, and the results in the first-operand location are unpredictable when destructive overlap exists. Operands are said to overlap destructively when the first-operand location is used as a source after data has been moved into it.

Operands do not overlap destructively if the leftmost character of the first operand does not coincide with any of the third-operand characters participating in the other than the leftmost character of the third operand. When an operand wraps around from location \( 2^{24} - 1 \) (or \( 2^{31} - 1 \) or \( 2^{64} - 1 \)) to location 0, operand characters in locations up to and including \( 2^{24} - 1 \) (or \( 2^{31} - 1 \) or \( 2^{64} - 1 \)) are considered to be to the left of characters in locations from 0 up.

In the 24-bit addressing mode, wraparound is from location \( 2^{24} - 1 \) to location 0; in the 31-bit addressing mode, wraparound is from location \( 2^{31} - 1 \) to location 0; and, in the 64-bit addressing mode, wraparound is from location \( 2^{64} - 1 \) to location 0.

When the length specified in general register \( R_1 + 1 \) is zero, no movement takes place, and condition code 0 or 1 is set to indicate the relative values of the lengths.

For the nonpadding part of the operation, accesses to the operands for MOVE LONG UNICODE are single-access references. These accesses do not necessarily appear to occur in a left-to-right direction as observed by other CPUs and by channel programs. During the nonpadding part of the operation, operands appear to be accessed doubleword concurrent as observed by other CPUs, provided that both operands start on doubleword boundaries, are an integral number of doublewords in length, and do not overlap.

As observed by other CPUs and by channel programs, that portion of the first operand which is filled with the two-byte padding character is not necessarily stored into in a left-to-right direction and may appear to be stored into more than once.

At the completion of the operation, the length in general register \( R_1 + 1 \) is decremented by 2 times the number of characters stored at the first-operand location, and the address in general register \( R_1 \) is incremented by the same amount. The length in general register \( R_3 + 1 \) is decremented by 2 times the number of characters moved out of the third-operand location, and the address in general register \( R_3 \) is incremented by the same amount.

If the operation is completed because a CPU-determined number of characters have been moved without reaching the end of the first operand, the lengths in general registers \( R_1 + 1 \) and \( R_3 + 1 \) are decremented by 2 times the number of characters moved, and the addresses in general registers \( R_1 \) and \( R_3 \) are incremented by the same number, so that the instruction, when reexecuted, resumes at the next character to be moved.

If the operation is completed during padding, the length field in general register \( R_3 + 1 \) is zero, the address in general register \( R_3 \) is incremented by 2 times the number of characters moved from operand 3, and general registers \( R_1 \) and \( R_1 + 1 \) reflect the extent of the padding operation.
In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers \( R_{\text{one}} \), \( R_{\text{one}} + 1 \), \( R_{\text{two}} \), and \( R_{\text{two}} + 1 \), always remain unchanged.

The two-byte padding character may be formed from \( D_2(B_2) \) multiple times during the execution of the instruction, and the registers designated by \( R_{\text{one}} \) and \( R_{\text{three}} \) may be updated multiple times. Therefore, if \( B_2 \) equals \( R_{\text{one}} \), \( R_{\text{one}} + 1 \), \( R_{\text{three}} \), or \( R_{\text{three}} + 1 \) and is subject to change during the execution of the instruction, the results are unpredictable.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

At the completion of the operation in the 24-bit or 31-bit addressing mode, the leftmost bits which are not part of the address in bit positions 32-63 of general registers \( R_{\text{one}} \) and \( R_{\text{three}} \) may be set to zeros or may remain unchanged from their original values, including the case when one or both of the original length values are zeros.

When the length of an operand is zero, no access exceptions for that operand are recognized. Similarly, when the third operand is longer than the first operand, access exceptions are not recognized for the part of the third-operand field that is in excess of the first-operand field. For operands longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the current location being processed. Access exceptions are not recognized for an operand if the \( R \) field or length associated with that operand is odd. Also, when the \( R_{\text{one}} \) field or length is odd, PER storage-alteration events are not recognized, and no change bits are set.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All characters moved, operand lengths equal</td>
</tr>
<tr>
<td>1</td>
<td>All characters moved, first-operand length low</td>
</tr>
<tr>
<td>2</td>
<td>All characters moved, first-operand length high</td>
</tr>
<tr>
<td>3</td>
<td>CPU-determined number of characters moved without reaching end of first operand</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Access (fetch, operand 3; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

**Programming Notes:**

1. MOVE LONG UNICODE is intended for use in place of MOVE LONG or MOVE LONG EXTENDED when the padding character is two bytes. The character may be a Unicode character or any other double-byte character. MOVE LONG UNICODE sets condition code 3 in cases in which MOVE LONG would be interrupted.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the movement. The program need not determine the number of characters that were moved.

3. MOVE LONG UNICODE may be used for filling storage with padding characters by placing the padding character in the second-operand address and setting the third-operand length to zero. However, the stores associated with this clearing may be multiple-access stores and should not be used to clear an area if the possibility exists that another CPU or a channel program will attempt to access and use the area as soon as it appears to be zero. For more details, see "Storage-Operand Consistency" on page 5-98.

4. When the contents of the \( R_{\text{one}} \) and \( R_{\text{three}} \) fields are the same, the contents of the designated registers are incremented or decremented only by 2 times the number of characters moved, not by 4 times the number of characters moved. The condition code is finally set to 0 after possible settings to 3.

5. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

6. If padding with a Unicode space character is required (or any character whose representation is less than or equal to FFF hex), the character may be represented in the displacement field of the instruction, for example:

   \[ \text{MVCLU 6,8,X'020'} \]
The rightmost four bits of each byte in the second operand are placed in the rightmost bit positions of the corresponding bytes in the first operand. The leftmost four bits of each byte in the first operand remain unchanged.

Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2; fetch and store, operand 1)

**Programming Notes:**

1. An example of the use of the MOVE NUMERICs instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. MOVE NUMERICs moves the numeric portion of a decimal-data field that is in the zoned format. The zoned-decimal format is described in Chapter 8, "Decimal Instructions." The operands are not checked for valid sign and digit codes.

3. Accesses to the first operand of MOVE NUMERICs consist in fetching the rightmost four bits of each byte in the first operand and subsequently storing the updated value of the byte. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

All or part of the second operand is placed in the first-operand location. The operation proceeds until the end of the second operand is reached or a CPU-determined number of bytes have been moved, whichever occurs first. The CPU-determined number is at least one. The result is indicated in the condition code.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R1 and R2, respectively.

The handling of the addresses in general registers R1 and R2 is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R1 and R2 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The end of the second operand is indicated by an ending character in the last byte position of the operand. The ending character to be used to determine the end of the second operand is specified in bit positions 56-63 of general register 0. Bit positions 32-55 of general register 0 are reserved for possible future extensions and must contain all zeros; otherwise, a specification exception is recognized.

The operation proceeds left to right and ends as soon as the second-operand ending character has been moved or a CPU-determined number of second-operand bytes have been moved, whichever occurs first. The CPU-determined number is at least one. When the ending character is in the first byte position of the second operand, only the ending character is moved. When the ending character has been moved, condition code 1 is set. When a CPU-determined number of second-operand bytes not including an ending character...
have been moved, condition code 3 is set. Destructive overlap is not recognized. If the second operand is used as a source after it has been used as a destination, the results are unpredictable to the extent that an ending character in the second operand may not be recognized.

When condition code 1 is set, the address of the ending character in the first operand is placed in general register \( R_{1} \), and the contents of general register \( R_{2} \) remain unchanged. When condition code 3 is set, the address of the next byte to be processed in the first and second operands is placed in general registers \( R_{1} \) and \( R_{2} \), respectively. Whenever an address is placed in a general register, bits 32-39 of the register, in the 24-bit addressing mode, or bit 32, in the 31-bit addressing mode, are set to zeros. Bits 0-31 of the \( R_{1} \) and \( R_{2} \) registers always remain unchanged in the 24-bit or 31-bit mode.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

Access exceptions for the first and second operands are recognized only for that portion of the operand that is necessarily used in the operation.

The storage-operand-consistency rules are the same as for the MOVE (MVC) instruction, except that destructive overlap is not recognized.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Entire second operand moved; general register ( R_{1} ) updated with address of ending character in first operand; general register ( R_{2} ) unchanged</td>
</tr>
<tr>
<td>1</td>
<td>CPU-determined number of bytes moved; general registers ( R_{1} ) and ( R_{2} ) updated with addresses of next bytes</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Access (fetch, operand 2; store, operand 1)
- Specification

**Programming Notes:**

1. An example of the use of the MOVE STRING instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the data movement. The program need not determine the number of bytes that were moved.

3. \( R_{1} \) or \( R_{2} \) may be zero, in which case general register 0 is treated as containing an address and also the ending character.

4. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

**MOVE WITH OFFSET**

\[
\text{MVO } D_{1}(L_{1},B_{1}),D_{2}(L_{2},B_{2}) \quad [SS]
\]

\[
\begin{array}{cccccccc}
'F1' & L_{1} & L_{2} & B_{1} & D_{1} & B_{2} & D_{2} \\
0 & 8 & 12 & 16 & 20 & 32 & 36 & 47
\end{array}
\]

The second operand is placed to the left of and adjacent to the rightmost four bits of the first operand.

The rightmost four bits of the first operand are attached as the rightmost bits to the second operand, the second-operand bits are offset by four bit positions, and the result is placed at the first-operand location.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first operand is too short to contain all of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if the operands were processed one byte at a time, as if each result byte were stored immediately after fetching the necessary operand bytes, and as if the left digit of each second-operand byte were to remain available for the next result byte and need not be refetched.
**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2; fetch and store, operand 1)

**Programming Notes:**
1. An example of the use of the MOVE WITH OFFSET instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. MOVE WITH OFFSET may be used to shift packed decimal data by an odd number of digit positions. The packed-decimal format is described in Chapter 8, "Decimal Instructions." The operands are not checked for valid sign and digit codes. In many cases, however, SHIFT AND ROUND DECIMAL may be more convenient to use.

3. Access to the rightmost byte of the first operand of MOVE WITH OFFSET consists in fetching the rightmost four bits and subsequently storing the updated value of this byte. These fetch and store accesses to the rightmost byte of the first operand do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (O1) in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

4. The storage-operand references for MOVE WITH OFFSET may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

**MOVE ZONES**

```
MVZ D1(L,B1),D2(B2) [SS]
```

| 'D3' | L | B1 | B2 | D1 | D2 |
| 0    | 8 | 16 | 20 | 32 | 36 | 47 |

The leftmost four bits of each byte in the second operand are placed in the leftmost four bit positions of the corresponding bytes in the first operand. The rightmost four bits of each byte in the first operand remain unchanged.

Each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after the necessary operand byte is fetched.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2; fetch and store, operand 1)

**Programming Notes:**
1. An example of the use of the MOVE ZONES instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. MOVE ZONES moves the zoned portion of a decimal field in the zoned format. The zoned format is described in Chapter 8, "Decimal Instructions." The operands are not checked for valid sign and digit codes.

3. Accesses to the first operand of MOVE ZONES consist in fetching the leftmost four bits of each byte in the first operand and subsequently storing the updated value of the byte. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for the OR (O1) instruction in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples."

**MULTIPLY**

```
MR R1,R2 [RR]
```

| '1C' | R1 | R2 |
| 0    | 8  | 12 | 15 |
The 32-bit first operand (the multiplicand) is multiplied by the 32-bit second operand (the multiplier), and the 64-bit product is placed at the first-operand location.

The R₁ field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

Both the multiplicand and multiplier are treated as 32-bit signed binary integers. The multiplicand is in bit positions 32-63 of general register R₁ + 1. For MULTIPLY (MR), the multiplier is in bit positions 32-63 of general register R₂. The contents of general register R₁ and of bit positions 0-31 of general register R₁ + 1 and, for MR, of general register R₂ are ignored.

The product is a 64-bit signed binary integer. Bits 0-31 of the product replace bits 32-63 of general register R₁. Bits 32-63 of the product replace bits 32-63 of general register R₁ + 1. Bits 0-31 of general registers R₁ and R₁ + 1 remain unchanged. An overflow cannot occur.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of M only)
- Specification

**Programming Notes:**

1. An example of the use of the MULTIPLY instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The significant part of the product usually occupies 62 bit positions or fewer. Only when two maximum 32-bit negative numbers are multiplied are 63 significant product bits formed.

---

**MULTIPLY HALFWORD**

MH R₁,D₂(X₂,B₂) [RX]

<table>
<thead>
<tr>
<th>'5C'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The 32-bit or 64-bit first operand (the multiplicand) is multiplied by the 16-bit second operand (the multiplier), and the rightmost 32 or 64 bits of the product are placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer.

For MULTIPLY HALFWORD and MULTIPLY HALFWORD IMMEDIATE (MHI), the multiplicand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R₁, and it is replaced by the rightmost 32 bits of the signed-binary-integer product. The bits to the left of the 32 rightmost bits of the product are not tested for significance; no overflow indication is given. Bits 0-31 of general register R₁ are ignored and remain unchanged.

For MULTIPLY HALFWORD IMMEDIATE (MGHI), The multiplicand is treated as a 64-bit signed binary integer in bit positions 0-63 of general register R₁, and it is replaced by the rightmost 64 bits of the signed-binary-integer product. The bits to the left of the 64 rightmost bits of the product are not tested for significance; no overflow indication is given.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.
**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of MH only)

**Programming Notes:**
1. An example of the use of the MULTIPLY HALFWORD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. For MH and MHI, the significant part of the product usually occupies 46 bit positions or fewer. Only when two maximum negative numbers are multiplied are 47 significant product bits formed. Since the rightmost 32 bits of the product are placed unchanged at the first-operand location, ignoring all bits to the left, the sign bit of the result may differ from the true sign of the product in the case of overflow. For a negative product, the 32 bits placed in register R₁ are the rightmost part of the product in two's-complement notation. For MGHI, the significant part of the product usually occupies 78 bit positions or fewer.

### MULTIPLY LOGICAL

**MLR**  \( R₁, R₂ \) [RRE]

<table>
<thead>
<tr>
<th></th>
<th>'B996'</th>
<th>////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

**MLGR**  \( R₁, R₂ \) [RRE]

<table>
<thead>
<tr>
<th></th>
<th>'B986'</th>
<th>////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

**ML**  \( R₁, D₂(\text{X₂,B₂}) \) [RXY]

<table>
<thead>
<tr>
<th></th>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₀</th>
<th>DH₀</th>
<th>'96'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
<td>47</td>
</tr>
</tbody>
</table>

**MLG**  \( R₁, D₂(\text{X₂,B₂}) \) [RXY]

<table>
<thead>
<tr>
<th></th>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₀</th>
<th>DH₀</th>
<th>'86'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
<td>47</td>
</tr>
</tbody>
</table>

The 32-bit or 64-bit first operand (the multiplicand) is multiplied by the 32-bit or 64-bit second operand (the multiplier), and the 64-bit or 128-bit product is placed at the first-operand location.

The \( R₁ \) field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

For MULTIPLY LOGICAL (MLR, ML), both the multiplicand and the multiplier are treated as 32-bit unsigned binary integers. The multiplicand is in bit positions 32-63 of general register \( R₁ + 1 \). For MULTIPLY LOGICAL (MLR), the multiplier is in bit positions 32-63 of general register \( R₂ \). The contents of general register \( R₁ \) and of bit positions 0-31 of general register \( R₁ + 1 \) and, for MLR, of general register \( R₂ \) are ignored. The product is a 64-bit unsigned binary integer. Bits 0-31 of the product replace bits 32-63 of general register \( R₁ \), and bits 32-63 of the product replace bits 32-63 of general register \( R₁ + 1 \). Bits 0-31 of general registers \( R₁ \) and \( R₁ + 1 \) remain unchanged. An overflow cannot occur.

For MULTIPLY LOGICAL (MLGR, MLG), the multiplicand and the multiplier are treated as 64-bit unsigned binary integers. The multiplicand is in general register \( R₁ + 1 \). The contents of general register \( R₁ \) are ignored. The product is a 128-bit unsigned binary integer. Bits 0-63 of the product replace the contents of general register \( R₁ \), and bits 64-127 of the product replace the contents of general register \( R₁ + 1 \). An overflow cannot occur.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of ML and MLG only)
- Specification

### MULTIPLY SINGLE

**MSR**  \( R₁, R₂ \) [RRE]

<table>
<thead>
<tr>
<th></th>
<th>'B252'</th>
<th>////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>
For MULTIPLY SINGLE (MSGR, MSGFR, MSG, MSGF), the multiplicand, multiplier, and product are treated as 64-bit signed binary integers, except that, for MSGFR and MSGF, the multiplier is treated as a 32-bit signed binary integer. The multiplicant is taken from general register \( R_1 \) and is replaced by the rightmost 64 bits of the signed-binary-integer product. For MSGFR, the multiplier is in bit positions 32-63 of general register \( R_2 \). The bits to the left of the 64 rightmost bits of the product are not tested for significance; no overflow indication is given.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

The displacement for MS is treated as a 12-bit unsigned binary integer. The displacement for MSY, MSG, and MSGF is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of MS, MSY, MSG, MSGF only)
- Operation (MSY, if the long-displacement facility is not installed)

**OR**

\[
\text{OR } R_1, R_2 \quad [\text{RR}]
\]

\[
\begin{array}{ccc}
16 & R_1 & R_2 \\
0 & 8 & 12 & 15
\end{array}
\]

**OGR**

\[
\text{OGR } R_1, R_2 \quad [\text{RRE}]
\]

\[
\begin{array}{cccc}
'B981' & ////////// & R_1 & R_2 \\
0 & 16 & 24 & 28 & 31
\end{array}
\]

\[
\begin{array}{cccc}
'56' & R_1 & X_2 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 31
\end{array}
\]

The first operand (multiplicand) is multiplied by the second operand (multiplier), and the rightmost 32 or 64 bits of the product are placed at the first-operand location.

The sign of the product is determined by the rules of algebra from the multiplier and multiplicand sign, except that a zero result is always positive.

The displacement for MS is treated as a 12-bit unsigned binary integer. The displacement for MSY, MSG, and MSGF is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of MS, MSY, MSG, MSGF only)
- Operation (MSY, if the long-displacement facility is not installed)
The OR of the first and second operands is placed at the first-operand location.

The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero.

For OR (OC), each operand is processed left to right. When the operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after fetching the necessary operand bytes.

For OR (OI, OIY), the first operand is one byte in length, and only one byte is stored.

For OR (OR, O, OY), the operands are 32 bits, and for OR (OGR, OG), they are 64 bits.

The displacements for O, OI, and both operands of OC are treated as 12-bit unsigned binary integers. The displacement for OY, OIY, and OG is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

0  Result zero
1  Result not zero
2  --
3  --

**Program Exceptions:**

- Access (fetch, operand 2, O, OY, OG, and OC; fetch and store, operand 1, OI, OIY, and OC)
- Operation (OY and OIY, if the long-displacement facility is not installed)

**Programming Notes:**

1. Examples of the use of the OR instruction are given in [Appendix A, “Number Representation and Instruction-Use Examples.”](#)
2. OR may be used to set a bit to one.
3. Accesses to the first operand of OR (OI) and OR (OC) consist in fetching a first-operand byte from storage and subsequently storing the updated value. These fetch and store accesses to a particular byte do not necessarily occur one immediately after the other. Thus, OR cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown in [Multiprogramming and Multiprocessing Examples](#) in Appendix A, “Number Representation and Instruction-Use Examples.”
The second operand is ORed with bits of the first operand, and the result replaces those bits of the first operand. The remainder of the first operand remains unchanged.

For each instruction, the bits of the first operand that are ORed with the second operand and then replaced are as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bits ORed and Replaced</th>
</tr>
</thead>
<tbody>
<tr>
<td>OIHH</td>
<td>0-15</td>
</tr>
<tr>
<td>OIHL</td>
<td>16-31</td>
</tr>
<tr>
<td>OILH</td>
<td>32-47</td>
</tr>
<tr>
<td>OILL</td>
<td>48-63</td>
</tr>
</tbody>
</table>

The connective OR is applied to the operands bit by bit. The contents of a bit position in the result are set to one if the corresponding bit position in one or both operands contains a one; otherwise, the result bit is set to zero.

Resulting Condition Code:

0  Sixteen-bit result zero
1  Sixteen-bit result not zero
2  --
3  --

Program Exceptions: None.

PACK

PACK  \[ \text{D1(L1,B1),D2(L2,B2)} \]  [SS]

The format of the second operand is changed from zoned to packed, and the result is placed at the first-operand location. The zoned and packed formats are described in Chapter 8, “Decimal Instructions.”

The second operand is treated as having the zoned format. The numeric bits of each byte are treated as a digit. The zone bits are ignored, except the zone bits in the rightmost byte, which are treated as a sign.

The sign and digits are moved unchanged to the first operand and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first operand is too short to contain all digits of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if each result byte were stored immediately after fetching the necessary operand bytes. Two second-operand bytes are needed for each result byte, except for the rightmost byte of the result field, which requires only the rightmost second-operand byte.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
**Programming Notes:**

1. An example of the use of the PACK instruction is given in [Appendix A, "Number Representation and Instruction-Use Examples."](#)

2. PACK may be used to interchange the two hexadecimal digits in one byte by specifying a zero in the \( L_1 \) and \( L_2 \) fields and the same address for both operands.

3. To remove the zone bits of all bytes of a field, including the rightmost byte, both operands should be extended on the right with a dummy byte, which subsequently should be ignored in the result field.

4. The storage-operand references for PACK may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

**PACK ASCII**

<table>
<thead>
<tr>
<th>'E9'</th>
<th>( L_2 )</th>
<th>( B_1 )</th>
<th>( D_1 )</th>
<th>( B_2 )</th>
<th>( D_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The format of the second operand is changed from ASCII to packed, and the result is placed at the first-operand location. The packed format is described in Chapter 8, "Decimal Instructions."

The second-operand bytes are treated as containing decimal digits, having the binary encoding 0000-1001 for 0-9, in their rightmost four bit positions. The leftmost four bit positions of a byte are ignored. The second operand is considered to be positive.

The implied positive sign (1100 binary) and the source digits are placed at the first-operand location. The source digits are moved unchanged and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros.

The length of the first operand is 16 bytes.

The length of the second operand is designated by the contents of the \( L_2 \) field. The second-operand length must not exceed 32 bytes (\( L_2 \) must be less than or equal to 31); otherwise, a specification exception is recognized.

When the length of the second operand is 32 bytes, the leftmost byte is ignored.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first-operand location is not necessarily stored into in any particular order.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

**Programming Notes:**

1. Although PACK ASCII is primarily intended to change the format of ASCII decimal digits, its use is not restricted to ASCII since the leftmost four bits of each byte are ignored.

2. The following example illustrates the use of the instruction to pack ASCII digits:

```
ASDIGITS  DS  OCL31
DC  X'3132333435'
DC  X'3637383930'
DC  X'3132333435'
DC  X'3637383930'
DC  X'3132333435'
DC  X'3637383930'
DC  X'31'  
PKDIGITS  DS  PL16
PKA  PKDIGITS,ASDIGITS(31)
```

3. The instruction can also be used to pack EBCDIC digits, which is especially useful when the length of the second operand is greater than the 16-byte second-operand limit of PACK.
4. The storage-operand references for PACK ASCII may be multiple-access references.
(See "Storage-Operand Consistency" on page 5-98.)

PACK UNICODE

The format of the second operand is changed from Unicode to packed, and the result is placed at the first-operand location. The packed format is described in Chapter 8, “Decimal Instructions.”

The two-byte second-operand characters are treated as Unicode Basic Latin characters containing decimal digits, having the binary encoding 0000-1001 for 0-9, in their rightmost four bit positions. The leftmost 12 bit positions of a character are ignored. The second operand is considered to be positive.

The implied positive sign (1100 binary) and the source digits are placed at the first-operand location. The source digits are moved unchanged and are not checked for valid codes. The sign is placed in the rightmost four bit positions of the rightmost byte of the result field, and the digits are placed adjacent to the sign and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros.

The length of the first operand is 16 bytes.

The byte length of the second operand is designated by the contents of the L2 field. The second-operand length must not exceed 32 characters or 64 bytes, and the byte length must be even (L2 must be less than or equal to 63 and must be odd); otherwise, a specification exception is recognized.

When the length of the second operand is 32 characters (64 bytes), the leftmost character is ignored.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first-operand location is not necessarily stored into in any particular order.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

**Programming Notes:**

1. The following example illustrates the use of PACK UNICODE to pack European numbers:

2. Because the leftmost 12 bits of each character are ignored, those Unicode decimal digits where the digit zero has four rightmost zero bits can also be packed by the instruction. For example, for Thai digits:
3. The storage-operand references for PACK
UNICODE may be multiple-access references.
(See "Storage-Operand Consistency" on
page 5-98.)

PERFORM LOCKED OPERATION

<table>
<thead>
<tr>
<th>PLO R1,D2(B2),R3,D4(B4)</th>
<th>[SS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EE' R1 R3 B2 D2 B4 D4</td>
<td></td>
</tr>
<tr>
<td>0 8 12 16 20 32 36 47</td>
<td></td>
</tr>
</tbody>
</table>

After the lock specified in general register 1 has been obtained, the operation specified by the function code in general register 0 is performed, and then the lock is released. However, as observed by other CPUs: (1) storage operands, including fields in a parameter list that may be used, may be fetched, and may be tested for store-type access exceptions if a store at a tested location is possible, before the lock is obtained, and (2) operands may be stored in the parameter list after the lock has been released. If an operand not in the parameter list is fetched before the lock is obtained, it is fetched again after the lock has been obtained.

The function code can specify any of six operations: compare and load, compare and swap, double compare and swap, and store, compare and swap and double store, or compare and swap and triple store.

A test bit in general register 0 specifies, when one, that a lock is not to be obtained and none of the six operations is to be performed but, instead, the validity of the function code is to be tested. This will be useful if additional function codes for additional operations are assigned in the future. This definition is written as if the test bit is zero except when stated otherwise.

If compare and load is specified, the first-operand comparison value and the second operand are compared. If they are equal, the fourth operand is placed in the third-operand location. If the comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value.

If compare and swap is specified, the first-operand comparison value and the second operand are compared. If they are equal, the first-operand replacement value is stored at the second-operand location. If the comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value.

If double compare and swap is specified, the first-operand comparison value and the second operand are compared. If they are equal, the third-operand comparison value and the fourth operand are compared. If both comparisons indicate equality, the first-operand and third-operand replacement values are stored at the second-operand location and fourth-operand location, respectively. If the first comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value. If the first comparison indicates equality but the second does not, the fourth operand is placed in the third-operand-comparison-value location as a new third-operand comparison value.

If compare and swap and store, double store, or triple store is specified, the first-operand comparison value and the second operand are compared. If they are equal, the first-operand replacement value is stored at the second-operand location, and the third operand is stored at the fourth-operand location. Then, if the operation is the double-store or triple-store operation, the fifth operand is stored at the sixth-operand location, and, if it is the triple-store operation, the seventh operand is stored at the eighth-operand location. If the first-operand comparison indicates inequality, the second operand is placed in the first-operand-comparison-value location as a new first-operand comparison value.

After any of the six operations, the result of the comparison or comparisons is indicated in the condition code.
The function code (FC) is in bit positions 56-63 of general register 0. The function code specifies not only the operation to be performed but also the length of the operands and whether the first-operand comparison and replacement values and the third operand or third-operand comparison and replacement values, which are referred to collectively simply as the first and third operands, are in general registers or a parameter list. The pattern of the function codes is as follows:

- A function code that is a multiple of 4 (including 0) specifies a 32-bit length with the first and third operands in bit positions 32-63 of general registers.
- A function code that is one more than a multiple of 4 specifies a 64-bit length with the first and third operands in a parameter list.
- A function code that is 2 more than a multiple of 4 specifies a 64-bit length with the first and third operands in bit positions 0-63 of general registers.
- A function code that is 3 more than a multiple of 4 specifies a 128-bit length with the first and third operands in a parameter list.

Figure 7-75 shows the function codes, operation names, and operand lengths, and also symbols that may be used to refer to the operations in discussions. For example, PLO.DCS may be used to mean PERFORM LOCKED OPERATION with function code 8. In the symbols, the letter “G” indicates a 64-bit operand length, the letter “R” indicates that some or all 64-bit operands are in general registers, and the letter “X” indicates a 128-bit operand length.

The CPU can perform all of the operations specified by the function codes listed in Figure 7-75. Function codes specifying operations that the CPU can perform are called valid. Function codes that have not been assigned to operations or that specify operations that the CPU cannot perform because the operations are not implemented (installed) are called invalid.

Bit 55 of general register 0 is the test bit (T). When bit 55 is zero, the function code in general register 0 must be valid; otherwise, a specification exception is recognized. When bit 55 is one, the condition code is set to 0 if the function code is valid or to 3 if the function code is invalid, and no other operation is performed.

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Operation</th>
<th>Operand Length (Bits)</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Compare and load</td>
<td>32</td>
<td>CL</td>
</tr>
<tr>
<td>1</td>
<td>Same as 0</td>
<td>64</td>
<td>CLG</td>
</tr>
<tr>
<td>2</td>
<td>Same as 0</td>
<td>64</td>
<td>CLGR</td>
</tr>
<tr>
<td>3</td>
<td>Same as 0</td>
<td>128</td>
<td>CLX</td>
</tr>
<tr>
<td>4</td>
<td>Compare and swap</td>
<td>32</td>
<td>CS</td>
</tr>
<tr>
<td>5</td>
<td>Same as 4</td>
<td>64</td>
<td>CSG</td>
</tr>
<tr>
<td>6</td>
<td>Same as 4</td>
<td>64</td>
<td>CSGR</td>
</tr>
<tr>
<td>7</td>
<td>Same as 4</td>
<td>128</td>
<td>CSX</td>
</tr>
<tr>
<td>8</td>
<td>Double compare and swap</td>
<td>32</td>
<td>DCS</td>
</tr>
<tr>
<td>9</td>
<td>Same as 8</td>
<td>64</td>
<td>DCSG</td>
</tr>
<tr>
<td>10</td>
<td>Same as 8</td>
<td>64</td>
<td>DCSGR</td>
</tr>
<tr>
<td>11</td>
<td>Same as 8</td>
<td>128</td>
<td>DCSX</td>
</tr>
<tr>
<td>12</td>
<td>Compare and swap and store</td>
<td>32</td>
<td>CSST</td>
</tr>
<tr>
<td>13</td>
<td>Same as 12</td>
<td>64</td>
<td>CSSTG</td>
</tr>
<tr>
<td>14</td>
<td>Same as 12</td>
<td>64</td>
<td>CSSTGR</td>
</tr>
<tr>
<td>15</td>
<td>Same as 12</td>
<td>128</td>
<td>CSSTX</td>
</tr>
<tr>
<td>16</td>
<td>Compare and swap and double store</td>
<td>32</td>
<td>CSDST</td>
</tr>
<tr>
<td>17</td>
<td>Same as 16</td>
<td>64</td>
<td>CSDSTG</td>
</tr>
<tr>
<td>18</td>
<td>Same as 16</td>
<td>64</td>
<td>CSDSTGR</td>
</tr>
<tr>
<td>19</td>
<td>Same as 16</td>
<td>128</td>
<td>CSDSTX</td>
</tr>
<tr>
<td>20</td>
<td>Compare and swap and triple store</td>
<td>32</td>
<td>CSTST</td>
</tr>
<tr>
<td>21</td>
<td>Same as 20</td>
<td>64</td>
<td>CSTSTG</td>
</tr>
<tr>
<td>22</td>
<td>Same as 20</td>
<td>64</td>
<td>CSTSTGR</td>
</tr>
<tr>
<td>23</td>
<td>Same as 20</td>
<td>128</td>
<td>CSTSTX</td>
</tr>
</tbody>
</table>

Figure 7-75. PERFORM LOCKED OPERATION Function Codes and Operations

Bits 32-54 of general register 0 must be all zeros; otherwise, a specification exception is recognized. When bit 55 of the register is one, this is the only exception that can be recognized. Bits 0-31 of general register 0 are ignored.
The lock to be used is represented by a program lock token (PLT) whose logical address is specified in general register 1. In the 24-bit addressing mode, the PLT address is bits 40-63 of general register 1, and bits 0-39 of the register are ignored. In the 31-bit addressing mode, the PLT address is bits 33-63 of the register, and bits 0-32 of the register are ignored. In the 64-bit addressing mode, the PLT address is bits 0-63 of the register.

The contents of general registers 0 and 1 described above are as follows:

For the even-numbered function codes, including 0, the first-operand comparison value is in general register R1. For the even-numbered function codes beginning with 4, the first-operand replacement value is in general register R1 + 1, and R1 designates an even-odd pair of registers and must designate an even-numbered register; otherwise, a specification exception is recognized. For function codes 0 and 2, R1 can be even or odd.

For function codes 0, 2, 12, and 14, the third operand is in general register R3, and R3 can be even or odd.

For function codes 8 and 10, the third-operand comparison value is in general register R3, the third-operand replacement value is in general register R3 + 1, and R3 designates an even-odd pair of registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

For function codes 0, 2, 8, 10, 12, and 14, the B4 and D4 fields of the instruction specify the fourth-operand address.

For function codes 1, 3, 5, 7, 9, 11, 13, 15, and 16-23, the B4 and D4 fields of the instruction specify the address of a parameter list that is used by the instruction, and this address is not called the fourth-operand address. The parameter list contains odd-numbered operands, including comparison and replacement values, and addresses of even-numbered operands other than the second operand. In the access-register mode, the parameter list also contains access-list-entry tokens (ALETs) associated with the even-numbered-operand addresses.

In the access-register mode, for function codes that cause use of a parameter list containing an ALET, R3 must not be zero; otherwise, a specification exception is recognized.

The rules about R1 and R3, and the use of the address specified by B4 and D4, are summarized in Figure 7-76 on page 7-159.

For all function codes, the B2 and D2 fields of the instruction specify the second-operand address.

For function codes 0, 2, 8, 10, 12, and 14, the B4 and D4 fields of the instruction specify the fourth-operand address.

For function codes 1, 3, 5, 7, 9, 11, 13, 15, and 16-23, the B4 and D4 fields of the instruction specify the address of a parameter list that is used by the instruction, and this address is not called the fourth-operand address. The parameter list contains odd-numbered operands, including comparison and replacement values, and addresses of even-numbered operands other than the second operand. In the access-register mode, the parameter list also contains access-list-entry tokens (ALETs) associated with the even-numbered-operand addresses.

In the access-register mode, for function codes that cause use of a parameter list containing an ALET, R3 must not be zero; otherwise, a specification exception is recognized.

The rules about R1 and R3, and the use of the address specified by B4 and D4, are summarized in Figure 7-76 on page 7-159.

For the even-numbered function codes, including 0, the first-operand comparison value is in general register R1. For the even-numbered function codes beginning with 4, the first-operand replacement value is in general register R1 + 1, and R1 designates an even-odd pair of registers and must designate an even-numbered register; otherwise, a specification exception is recognized. For function codes 0 and 2, R1 can be even or odd.

For function codes 0, 2, 12, and 14, the third operand is in general register R3, and R3 can be even or odd.

For function codes 8 and 10, the third-operand comparison value is in general register R3, the third-operand replacement value is in general register R3 + 1, and R3 designates an even-odd pair of registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

For function codes 0, 2, 8, 10, 12, and 14, the B4 and D4 fields of the instruction specify the fourth-operand address.

For function codes 1, 3, 5, 7, 9, 11, 13, 15, and 16-23, the B4 and D4 fields of the instruction specify the address of a parameter list that is used by the instruction, and this address is not called the fourth-operand address. The parameter list contains odd-numbered operands, including comparison and replacement values, and addresses of even-numbered operands other than the second operand. In the access-register mode, the parameter list also contains access-list-entry tokens (ALETs) associated with the even-numbered-operand addresses.

In the access-register mode, for function codes that cause use of a parameter list containing an ALET, R3 must not be zero; otherwise, a specification exception is recognized.

The rules about R1 and R3, and the use of the address specified by B4 and D4, are summarized in Figure 7-76 on page 7-159.
<table>
<thead>
<tr>
<th>Function Codes</th>
<th>Operation</th>
<th>R₃</th>
<th>R₄</th>
<th>D₄(B₄)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 and 2</td>
<td>Compare and load</td>
<td>EO</td>
<td>EO</td>
<td>Op₄a</td>
</tr>
<tr>
<td>1 and 3</td>
<td>Compare and load</td>
<td>-NZ</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>4 and 6</td>
<td>Compare and swap</td>
<td>E</td>
<td>-</td>
<td>PLA</td>
</tr>
<tr>
<td>5 and 7</td>
<td>Compare and swap</td>
<td>-</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>8 and 10</td>
<td>Double compare and swap</td>
<td>E</td>
<td>E</td>
<td>Op₄a</td>
</tr>
<tr>
<td>9 and 11</td>
<td>Double compare and swap</td>
<td>-NZ</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>12 and 14</td>
<td>Compare and swap and store</td>
<td>E</td>
<td>EO</td>
<td>Op₄a</td>
</tr>
<tr>
<td>13 and 15</td>
<td>Compare and swap and store</td>
<td>-NZ</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>16 and 18</td>
<td>Compare and swap and double store</td>
<td>E</td>
<td>NZ</td>
<td>PLA</td>
</tr>
<tr>
<td>17 and 19</td>
<td>Compare and swap and double store</td>
<td>-NZ</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>20 and 22</td>
<td>Compare and swap and triple store</td>
<td>E</td>
<td>NZ</td>
<td>PLA</td>
</tr>
<tr>
<td>21 and 23</td>
<td>Compare and swap and triple store</td>
<td>-NZ</td>
<td>PLA</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**
- Ignored.
- E Must be even.
- NZ Must be nonzero in the access-register mode. Ignored otherwise.
- Op₄a D₄(B₄) is operand-4 address.
- PLA D₄(B₄) is parameter-list address.

Figure 7-76. Register Rules and D₄(B₄) Usage for PERFORM LOCKED OPERATION

Figure 7-77 on page 7-160 shows the locations of the operands (including operand comparison and replacement values), operand addresses, and parameter-list address used by the instruction.

Operand addresses in a parameter list, if used, are in doublewords in the list. In the 24-bit addressing mode, an operand address is bits 40-63 of a doubleword, and bits 0-39 of the doubleword are ignored. In the 31-bit addressing mode, an operand address is bits 33-63 of a doubleword, and bits 0-32 of the doubleword are ignored. In the 64-bit addressing mode, an operand address is bits 0-63 of a doubleword.

In the access-register mode, access register 1 specifies the address space containing the program lock token (PLT), access register B₂ specifies the address space containing the second operand, and access register B₄ specifies the address space containing a fourth operand or a parameter list as shown in Figure 7-77 on page 7-160. Also, for an operand whose address is in the parameter list, an access-list-entry token (ALET) is in the list along with the address and is used in the access-register mode to specify the address space containing the operand.

In the access-register mode, if an access exception or PER storage-alteration event is recognized for an operand whose address is in the parameter list, the associated ALET in the parameter list is loaded into access register R₃ when the exception or event is recognized. Then, during the resulting program interruption, if a value is due to be stored as the exception access identification at real location 160 or the PER access identification at real location 161, R₃ is stored. If the instruction execution is completed without the recognition of an exception or event, the contents of access register R₃ are unpredictable. When not in the access-register mode, or when a parameter list containing an ALET is not used, the contents of access register R₃ remain unchanged.

The even-numbered (2, 4, 6, and 8) storage operands must be designated on an integral boundary, which is a word boundary for function codes that are a multiple of 4, a doubleword boundary for function codes that are one or 2 more than a multiple of 4, or a quadword boundary for function codes that are 3 more than a multiple of 4. A parameter list, if used, must be designated on a doubleword boundary. Otherwise, a specification exception is recognized. The program-lock-token (PLT) address in general register 1 does not have a boundary-alignment requirement.

All unused fields in a parameter list should contain all zeros; otherwise, the program may not operate compatibly in the future.

A serialization operation is performed immediately after the lock is obtained and again immediately before it is released. However, values fetched from the parameter list before the lock is obtained are not necessarily refetched. A serialization operation is not performed if the test bit, bit 55 of general register 0, is one.

In the following figures showing the parameter lists for the different function codes, the offsets shown on the left are byte values.
<table>
<thead>
<tr>
<th>Function Codes¹</th>
<th>Operation</th>
<th>Op1c</th>
<th>Op1r</th>
<th>Op2a</th>
<th>Op3 or Op3c</th>
<th>Op3r</th>
<th>Op4a</th>
<th>Op5 and Op6a</th>
<th>Op7 and Op8a</th>
<th>PLa</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 and 2</td>
<td>Compare and load</td>
<td>R₁</td>
<td>-</td>
<td>D₂(B₂)</td>
<td>R₃</td>
<td>D₄(B₄)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 and 3</td>
<td>Compare and load</td>
<td>PL</td>
<td>-</td>
<td>D₂(B₂)</td>
<td>PL</td>
<td>PL</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>D₄(B₄)</td>
</tr>
<tr>
<td>4 and 6</td>
<td>Compare and swap</td>
<td>R₁</td>
<td>R₁₊₁</td>
<td>D₂(B₂)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>5 and 7</td>
<td>Compare and swap</td>
<td>PL</td>
<td>PL</td>
<td>D₂(B₂)</td>
<td>R₃</td>
<td>R₃₊₁</td>
<td>D₄(B₄)</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>8 and 10</td>
<td>Double compare and swap</td>
<td>R₁</td>
<td>R₁₊₁</td>
<td>D₂(B₂)</td>
<td>R₃</td>
<td>R₃₊₁</td>
<td>D₄(B₄)</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>9 and 11</td>
<td>Double compare and swap</td>
<td>PL</td>
<td>PL</td>
<td>D₂(B₂)</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>-</td>
<td>-</td>
<td>D₄(B₄)</td>
</tr>
<tr>
<td>12 and 14</td>
<td>Compare and swap and store</td>
<td>R₁</td>
<td>R₁₊₁</td>
<td>D₂(B₂)</td>
<td>R₃</td>
<td>D₄(B₄)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>13 and 15</td>
<td>Compare and swap and store</td>
<td>PL</td>
<td>PL</td>
<td>D₂(B₂)</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>-</td>
<td>-</td>
<td>D₄(B₄)</td>
</tr>
<tr>
<td>16 and 18</td>
<td>Compare and swap and double store</td>
<td>R₁</td>
<td>R₁₊₁</td>
<td>D₂(B₂)</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>-</td>
<td>D₄(B₄)</td>
<td></td>
</tr>
<tr>
<td>17 and 19</td>
<td>Compare and swap and double store</td>
<td>PL</td>
<td>PL</td>
<td>D₂(B₂)</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>-</td>
<td>D₄(B₄)</td>
<td></td>
</tr>
<tr>
<td>20 and 22</td>
<td>Compare and swap and triple store</td>
<td>R₁</td>
<td>R₁₊₁</td>
<td>D₂(B₂)</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>D₄(B₄)</td>
<td></td>
</tr>
<tr>
<td>21 and 23</td>
<td>Compare and swap and triple store</td>
<td>PL</td>
<td>PL</td>
<td>D₂(B₂)</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>PL</td>
<td>D₄(B₄)</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

¹ For function codes that are a multiple of 4 (including 0) or one more than a multiple of 4, operands in general registers are in bit positions 32-63 of the registers, and bits 0-31 of the registers are ignored and remain unchanged. For function codes that are two more than a multiple of 4, operands in general registers are in bit positions 0-63 of the registers.

- Operand, value, or address is not used in the operation.
- OpNc Operand-N comparison value.
- OpNr Operand-N replacement value.
- OpNa Operand-N address.
- PL Operand, value, or address is in the parameter list.
- PLa Parameter-list address.

Figure 7-77. Operand and Address Locations for PERFORM LOCKED OPERATION

**Function Codes 0-3 (Compare and Load)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-77.
The parameter list used for function code 1 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8. Operand-1 Comparison Value</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>64. Operand-4 ALET</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>72. Operand-4 Address</td>
</tr>
</tbody>
</table>

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the third operand is replaced by the fourth operand, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

**Function Codes 4-7 (Compare and Swap)**

The locations of the operands and addresses used by the instruction are as shown in Figure 7-77 on page 7-160.

The parameter list used for function code 5 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8. Operand-1 Comparison Value</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24. Operand-1 Replacement Value</td>
</tr>
</tbody>
</table>

The parameter list used for function code 7 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8. Operand-1 Comp. Value (continued)</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>16. Operand-1 Replacement Value</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24. Operand-1 Repl. Value (continued)</td>
</tr>
</tbody>
</table>

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.
Function Codes 8-11 (Double Compare and Swap)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-77 on page 7-160.

The parameter list used for function code 9 has the following format:

Parameter List for Function Code 9

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Operand-1 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Operand-1 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Operand-3 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Operand-3 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>Operand-4 ALET</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>Operand-4 Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the third-operand comparison value is compared to the fourth operand. When the third-operand comparison value is equal to the fourth operand (after the first-operand comparison value has been found equal to the second operand), the first-operand replacement value is stored at the second-operand location, the third-operand replacement value is stored at the fourth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

When the third-operand comparison value is not equal to the fourth operand (after the first-operand comparison value has been found equal to the second operand), the third-operand comparison value is replaced by the fourth operand, and condition code 2 is set.

Function Codes 12-15 (Compare and Swap and Store)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-77 on page 7-160.

The parameter list used for function code 11 has the following format:

Parameter List for Function Code 11

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand-1 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Operand-1 Comp. Value (continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Operand-1 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Operand-1 Repl. Value (continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Operand-3 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Operand-3 Comp. Value (continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Operand-3 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Operand-3 Repl. Value (continued)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Operand-4 ALET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>Operand-4 Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The parameter list used for function code 13 has the following format:

Parameter List for Function Code 13

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand-1 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Operand-1 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Operand-1 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Operand-1 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Operand-3 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Operand-3 Comparison Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Operand-3 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Operand-3 Replacement Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Operand-4 ALET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>Operand-4 Address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The parameter list used for function code 15 has the following format:

Parameter List for Function Code 15

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand-1 Comparison Value</td>
</tr>
<tr>
<td>8</td>
<td>Operand-1 Comp. Value (continued)</td>
</tr>
<tr>
<td>16</td>
<td>Operand-1 Replacement Value</td>
</tr>
<tr>
<td>24</td>
<td>Operand-1 Repl. Value (continued)</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>Operand 3</td>
</tr>
<tr>
<td>56</td>
<td>Operand 3 (continued)</td>
</tr>
<tr>
<td>64</td>
<td>Operand-4 ALET</td>
</tr>
<tr>
<td>72</td>
<td>Operand-4 Address</td>
</tr>
</tbody>
</table>

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, the third operand is stored at the fourth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

Function Codes 16-19 (Compare and Swap and Double Store)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-77 on page 7-160.

The parameter list used for function code 16 has the following format:

Parameter List for Function Code 16

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Operand 3</td>
</tr>
<tr>
<td>64</td>
<td>Operand-4 ALET</td>
</tr>
<tr>
<td>72</td>
<td>Operand-4 Address</td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>Operand 5</td>
</tr>
<tr>
<td>96</td>
<td>Operand-6 ALET</td>
</tr>
<tr>
<td>104</td>
<td>Operand-6 Address</td>
</tr>
</tbody>
</table>
The parameter list used for function code 17 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>104</td>
</tr>
</tbody>
</table>

The parameter list used for function code 18 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 18</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>104</td>
</tr>
</tbody>
</table>

7-164 z/Architecture Principles of Operation
The parameter list used for function code 19 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>104</td>
</tr>
</tbody>
</table>

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, the third operand is stored at the fourth-operand location, the fifth operand is stored at the sixth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

Function Codes 20-23 (Compare and Swap and Triple Store)

The locations of the operands and addresses used by the instruction are as shown in Figure 7-77 on page 7-160.

The parameter list used for function code 20 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>136</td>
</tr>
</tbody>
</table>

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, the third operand is stored at the fourth-operand location, the fifth operand is stored at the sixth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.
The parameter list used for function code 21 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 21</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>136</td>
</tr>
</tbody>
</table>

The parameter list used for function code 22 has the following format:

<table>
<thead>
<tr>
<th>Parameter List for Function Code 22</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>4/zerodot</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>128</td>
</tr>
<tr>
<td>136</td>
</tr>
</tbody>
</table>
The parameter list used for function code 23 has the following format:

Parameter List for Function Code 23

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operand-1 Comparison Value</td>
</tr>
<tr>
<td>8</td>
<td>Operand-1 Comp. Value (continued)</td>
</tr>
<tr>
<td>16</td>
<td>Operand-1 Replacement Value</td>
</tr>
<tr>
<td>24</td>
<td>Operand-1 Repl. Value (continued)</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>Operand 3</td>
</tr>
<tr>
<td>64</td>
<td>Operand 3 (continued)</td>
</tr>
<tr>
<td>72</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td></td>
</tr>
</tbody>
</table>

The first-operand comparison value is compared to the second operand. When the first-operand comparison value is equal to the second operand, the first-operand replacement value is stored at the second-operand location, the third operand is stored at the fourth-operand location, the fifth operand is stored at the sixth-operand location, the seventh operand is stored at the eighth-operand location, and condition code 0 is set.

When the first-operand comparison value is not equal to the second operand, the first-operand comparison value is replaced by the second operand, and condition code 1 is set.

### Locking

A lock is obtained at the beginning of the operation and released at the end of the operation. The lock obtained is represented by a program lock token (PLT) whose logical address is specified in general register 1 as already described.

A PLT is a value produced by a model-dependent transformation of the PLT logical address. Depending on the model, the PLT may be derived directly from the PLT logical address or, when DAT is on, from the real address that results from transformation of the PLT logical address by DAT. If DAT is used, access-register translation (ART) precedes DAT in the access-register mode.

A PLT selects one of a model-dependent number of locks within the configuration. Programs being executed by different CPUs can be assured of specifying the same lock only by specifying PLT logical addresses that are the same and that can be transformed to the same real address by the different CPUs.

Since a model may or may not use ART and DAT when forming a PLT, access-exception conditions that can be encountered during ART and DAT may or may not be recognized as exceptions. There is no accessing of a location designated by a PLT, but an addressing exception may be recognized for the location. A protection exception is not recognized for any reason during processing of a PLT logical address.

The CPU can hold one lock at a time.

When PERFORM LOCKED OPERATION is executed by this CPU and is to use a lock that is already held by another CPU due to the execution of a PERFORM LOCKED OPERATION instruction by the other CPU, the execution by this CPU is delayed until the lock is no longer held. An excessive delay can be caused only by a machine malfunction and is a machine-check condition.

The order in which multiple requests for the same lock are satisfied is undefined.

A nonrecoverable failure of a CPU while holding a lock may result in a machine check, entry into the check-stop state, or system check stop. The machine check is processing backup if all operands are undamaged or processing damage if...
register operands are damaged. If a machine check or the check-stop state is the result, either no storage operands have been changed or else all storage operands that were due to be changed have been correctly changed, and, in either case, the lock has been released. If the storage operands are not in either their correct original state or their correct final state, the result is system check stop.

Storage-Operand References

The accesses to the even-numbered storage operands appear to be word concurrent, as observed by other CPUs, for function codes that are a multiple of 4 or doubleword concurrent for function codes that are one, 2, or 3 more than a multiple of 4. The accesses to the doublewords in the parameter list appear to be doubleword concurrent, as observed by other CPUs, regardless of the function code.

As observed by other CPUs, all storage operands may be tested for access exceptions before a lock is obtained. (A channel program cannot observe a lock.)

As observed by other CPUs, in all operations except the compare-and-swap operation (which does not have a fourth operand), the fourth operand is accessed while the lock is held only if a comparison of the first-operand comparison value to the second operand while the lock is held has indicated equality. In these operations, the fourth operand is accessed before the lock is held only if a comparison of the first-operand comparison value to the second operand has indicated equality and only if, when DAT is on, an INVALIDATE PAGE TABLE ENTRY instruction executed by another CPU after the fetch of the second operand will not be the cause of a page-translation exception recognized for the fourth operand, which it will if it sets to one the page-invalid bit in the page-table entry for the fourth operand when this CPU does not have a TLB entry corresponding to that page-table entry. In the compare-and-swap-and-double-store and compare-and-swap-and-triple-store operations, the sixth operand, and also the eighth operand in the triple-store operation, are treated the same as the fourth operand described above. The reason for this specification about INVALIDATE PAGE TABLE ENTRY is given in programming note 6 on page 7-169.

Provided that accessing of an operand is not prohibited as described in the preceding paragraph, store-type access exceptions may be recognized for the operand even when a store does not occur because of the results of a comparison. A storage-alteration PER event is recognized, and a change bit is set, only if a store occurs.

When a comparison is made between an operand comparison value and an operand before the lock is obtained and indicates inequality, the lock still is obtained. The condition code is set only as a result of a comparison made while the lock is held. When condition code 1 or 2 is set, the first-operand comparison value or third-operand comparison value is replaced only by means of a fetch of the second operand or fourth operand, respectively, made while the lock is held, as observed by other CPUs.

In those cases when a store is performed to the second-operand location and one or more of the fourth-, sixth-, and eighth-operand locations, the store to the second-operand location is always performed last, as observed by other CPUs and by channel programs.

Stores into the parameter list may be performed while the lock is held or after it has been released.

A serialization operation is performed immediately after the lock is obtained and again immediately before it is released. However, values fetched from the parameter list before the lock is obtained are not necessarily refetched. A serialization operation is not performed if the test bit, bit 55 of general register 0, is one.

Access exceptions may be recognized for parameter-list locations even when the locations are not required in the operation. The locations are those beginning at offset 0 and extending up through the last location defined for the function code used.

For the compare-and-load and compare-and-swap operations, the operation is suppressed on all addressing and protection exceptions.

When a nonrecoverable failure of a CPU while holding a lock results in a machine check or entry into the check-stop state, either no storage operands have been changed or else all storage oper-
ands that were due to be changed have been correctly changed. The latter may be accomplished by repeating stores that were performed successfully before the failure. Therefore, there may be two single-access store references (possibly the store part of an update reference and then a store reference) to the store-type operands, with the first value stored equal to the second value stored.

**Resulting Condition Code:**

When test bit is zero:

0 All comparisons equal; replacement value or values stored or loaded
1 First-operand comparison not equal; first-operand comparison value replaced
2 – (all operations except double compare and swap)
2 First-operand comparison equal but third-operand comparison not equal; third-operand comparison value replaced (double compare and swap)
3 –

When test bit is one:

0 Function code valid
1 –
2 –
3 Function code invalid

**Program Exceptions:**

- Access (for all function codes, fetch, except addressing and protection for PLT location, program lock token, model-dependent; for all function codes, fetch and store, operand 2; for odd-numbered function codes, fetch and store, parameter list; for function codes 16, 18, 20, and 22, fetch, parameter list; for function codes 0-3, fetch, operand 4; for function codes 8-11, fetch and store, operand 4; for function codes 12-23, store, operand 4; for function codes 16-23, store, operand 6; for function codes 20-23, store, operand 8)
- Specification

**Programming Notes:**

1. An example of the use of the PERFORM LOCKED OPERATION instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. When the contents of storage locations are changed by PERFORM LOCKED OPERATION instructions that are executed concurrently by different CPUs and that use the same lock, the changes to operands not in the parameter list will be completed by one of the CPUs before they are begun by the other CPU, depending on which CPU first obtains the lock.

3. The compare-and-swap functions of PERFORM LOCKED OPERATION are not performed by means of interlocked-update references. Concurrent store references by another CPU to the storage operands, even if they are interlocked-update references, will interfere unpredictably, in terms of the resulting register and storage contents, with the intended operation of PERFORM LOCKED OPERATION. All changes to the contents of the storage locations must be made by PERFORM LOCKED OPERATION instructions that use the same lock, if predictable storage results are to be obtained.

4. Because a nonrecoverable failure of a CPU while executing PERFORM LOCKED OPERATION may cause two stores of the same value to a store-type operand, a concurrent store made by another CPU to the same operand but not by executing PERFORM LOCKED OPERATION may be lost.

5. When programs in different address spaces are using the same lock when DAT is on, the programs must ensure that they are using PLT logical addresses that are the same and that will be translated to the same real address regardless of the address space in which a translation occurs. Otherwise, the programs may in fact use different locks.

6. The section “Storage-Operand References” on page 7-168 contains a specification concerning the INVALIDATE PAGE TABLE ENTRY (IPTE) instruction. The need for the specification is shown by the following example that is possible without the specification.

   a. CPU 1 begins to execute a PERFORM LOCKED OPERATION instruction with function code 8, which is referred to as PLO.DCS. Operand 2 is a location, Qtail, containing the address (the first-operand comparison value) of the last element, element X, on a queue, and operand 4 is a location in that element containing the
address (0, the third-operand comparison value) of the next (nonexisting) element on the queue. The purpose of the PLO instruction is to enqueue an element by placing the address of the element (the first-operand and third-operand replacement values) in both operand 2 and operand 4. With the lock not held, the PLO instruction fetches operand 2 and compares it, with an equal result, to the first-operand comparison value.

b. CPU 2 completely executes a PLO.DCS instruction to dequeue element X, which is the only element on the queue, from the queue. The PLO instruction stores 0 in Qtail and also in Qhead, which is a location containing the address of the first element on the queue. The program on CPU 2 processes the dequeued element and then invokes the freemain service of the control program to deallocate the storage containing the element. The freemain service uses IPTE to set the page-invalid bit to one in the page-table entry for the page containing element X. The IPTE instruction immediately sets the page-invalid bit to one, and then it waits for the signal that all other CPUs have cleared their TLBs of entries corresponding to the page.

c. CPU 1 attempts to fetch operand 4. CPU 1 does not have a TLB entry for the operand-4 page. CPU 1 signals CPU 2 that the CPU 2 IPTE instruction may proceed.

d. CPU 2 completes its IPTE instruction. The program on CPU 2 sets a software bit in the page-table entry to one to indicate that the page has been freemained and that, therefore, a reference to the page should result in presentation by the control program of an addressing exception to the program making the reference.

e. CPU 1 attempts to do DAT for operand 4 and sees that the page-invalid bit is one. CPU 1 performs a program interruption indicating a page-translation exception. The exception handler sees that the software bit indicating freemained is one, and it presents an addressing exception to the CPU 1 program, which causes an abend of the program.

If CPU 1 had had a TLB entry for the page, its PLO instruction would not have been interrupted, and the comparison of the first-operand comparison value to the second operand while the lock was held would indicate that CPU 2 had changed the second operand. The PLO instruction would set condition code 1. If CPU 1 did not have a TLB entry but IPTE could not set the page-invalid bit to one while CPU 1 was executing an instruction, CPU 1 could successfully translate the operand-4 address and, again, discover while the lock was held that operand 2 had changed. The case when operand 2 points to element X but the freemained bit for the element-X page is one is a programming error.

7. Figure 7-78 on page 7-171 summarizes the results of the operation.
### Function Codes 0-3 (Compare and Load)

<table>
<thead>
<tr>
<th>Cond Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1c=Op2</td>
<td>Op2 → Op1c</td>
</tr>
<tr>
<td>Op3c=Op4</td>
<td>Op4 → Op3</td>
</tr>
</tbody>
</table>

### Function Codes 4-7 (Compare and Swap)

<table>
<thead>
<tr>
<th>Cond Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1r → Op2</td>
<td>Op2 → Op1c</td>
</tr>
</tbody>
</table>

### Function Codes 8-11 (Double Compare and Swap)

<table>
<thead>
<tr>
<th>Cond Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1r → Op2</td>
<td>Op4 → Op3c</td>
</tr>
<tr>
<td>Op3r → Op4</td>
<td>Op2 → Op1c</td>
</tr>
</tbody>
</table>

### Function Codes 12-15 (Compare and Swap and Store)

<table>
<thead>
<tr>
<th>Cond Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1r → Op2</td>
<td>Op2 → Op1c</td>
</tr>
<tr>
<td>Op3 → Op4</td>
<td>Op5 → Op6</td>
</tr>
</tbody>
</table>

### Function Codes 16-19 (Compare and Swap and Double Store)

<table>
<thead>
<tr>
<th>Cond Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1r → Op2</td>
<td>Op2 → Op1c</td>
</tr>
<tr>
<td>Op3 → Op4</td>
<td>Op5 → Op6</td>
</tr>
<tr>
<td>Op7 → Op8</td>
<td>Op4 → Op3c</td>
</tr>
</tbody>
</table>

### Function Codes 20-23 (Compare and Swap and Triple Store)

<table>
<thead>
<tr>
<th>Cond Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op1r → Op2</td>
<td>Op2 → Op1c</td>
</tr>
<tr>
<td>Op3 → Op4</td>
<td>Op5 → Op6</td>
</tr>
<tr>
<td>Op7 → Op8</td>
<td>Op6 → Op4</td>
</tr>
</tbody>
</table>

### Explanation:

- Not applicable.
- OpNc Operand-N comparison value.
- OpNr Operand-N replacement value.

---

**Figure 7-78. Summary of PERFORM LOCKED OPERATION Results**

**ROTATE LEFT SINGLE LOGICAL**

RLL R1,R3,D2(B2) [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>DLz</th>
<th>DHz</th>
<th>'1D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

RLLG R1,R3,D2(B2) [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>DLz</th>
<th>DHz</th>
<th>'1C'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The 32-bit or 64-bit third operand is rotated left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register R3. For ROTATE LEFT SINGLE LOGICAL (RLL), bits 0-31 of general registers R1 and R3 remain unchanged.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be rotated. The remainder of the address is ignored.
For RLL, the first and third operands are in bit positions 32-63 of general registers \( R_1 \) and \( R_3 \), respectively. For RLLG, the operands are in bit positions 0-63 of the registers.

All 32 or 64 bits of the third operand participate in a left shift. Each bit shifted out of the leftmost bit position of the operand reenters in the rightmost bit position of the operand.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**SEARCH STRING**

SRST \( R_1, R_2 \) [RRE]

<table>
<thead>
<tr>
<th>'B25E'</th>
<th>/////</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28 31</td>
</tr>
</tbody>
</table>

The second operand is searched until a specified character is found, the end of the second operand is reached, or a CPU-determined number of bytes have been searched, whichever occurs first. The CPU-determined number is at least 256. The result is indicated in the condition code.

The location of the leftmost byte of the second operand is designated by the contents of general register \( R_2 \). The location of the first byte after the second operand is designated by the contents of general register \( R_1 \).

The handling of the addresses in general registers \( R_1 \) and \( R_2 \) is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers \( R_1 \) and \( R_2 \) constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

In the access-register mode, the address space containing the second operand is specified only by means of access register \( R_2 \). The contents of access register \( R_1 \) are ignored.

The character for which the search occurs is specified in bit positions 56-63 of general register 0. Bit positions 32-55 of general register 0 are reserved for possible future extensions and must contain all zeros; otherwise, a specification exception is recognized.

The operation proceeds left to right and ends as soon as the specified character has been found in the second operand, the address of the next second-operand byte to be examined equals the address in general register \( R_1 \), or a CPU-determined number of second-operand bytes have been examined, whichever occurs first. The CPU-determined number is at least 256. When the specified character is found, condition code 1 is set. When the address of the next second-operand byte to be examined equals the address in general register \( R_1 \), condition code 2 is set. When a CPU-determined number of second-operand bytes have been examined, condition code 3 is set. When the CPU-determined number of second-operand bytes have been examined and the address of the next second-operand byte is in general register \( R_1 \), it is unpredictable whether condition code 2 or 3 is set.

When condition code 1 is set, the address of the specified character found in the second operand is placed in general register \( R_1 \), and the contents of general register \( R_2 \) remain unchanged. When condition code 3 is set, the address of the next byte to be processed in the second operand is placed in general register \( R_2 \), and the contents of general register \( R_1 \) remain unchanged. When condition code 2 is set, the contents of general registers \( R_1 \) and \( R_2 \) remain unchanged. Whenever an address is placed in a general register, bits 32-39 of the register, in the 24-bit addressing mode, or bit 32, in the 31-bit addressing mode, are set to zeros. Bits 0-31 of the \( R_1 \) and \( R_2 \) registers always remain unchanged in the 24-bit or 31-bit mode.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

Access exceptions for the second operand are recognized only for that portion of the operand that is necessarily examined.
The storage-operand-consistency rules are the same as for the COMPARE LOGICAL LONG instruction.

**Resulting Condition Code:**

0  --  Specified character found; general register \( R_1 \) updated with address of character; general register \( R_2 \) unchanged

1  Specified character not found in entire second operand; general registers \( R_1 \) and \( R_2 \) unchanged

2  CPU-determined number of bytes searched; general register \( R_1 \) unchanged; general register \( R_2 \) updated with address of next byte

**Program Exceptions:**

- Access (fetch, operand 2)
- Specification

**Programming Notes:**

1. Examples of the use of the SEARCH STRING instruction are given in [Appendix A, “Number Representation and Instruction-Use Examples”](#).

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the search. The program need not determine the number of bytes that were searched.

3. When the address in general register \( R_1 \) equals the address in general register \( R_2 \), condition code 2 is set immediately, and access exceptions are not recognized. When the address in general register \( R_1 \) is less than the address in general register \( R_2 \), condition code 2 can be set only if the operand wraps around from the top of storage to location 0.

4. \( R_1 \) or \( R_2 \) may be zero, in which case general register 0 is treated as containing an address and also the specified character.

5. When it is desired to search a string of unknown length for its ending character, and assuming that (1) the string does not start below location 256 (or below location 1 if the ending character is 00 hex), (2) the string does not wrap around to location 0, and (3) the specified character in general register 0 need not be preserved, then \( R_1 \) can be zero in order to have SEARCH STRING use only two general registers instead of three.

**SEARCH STRING UNICODE**

\[
\begin{array}{ccc}
\text{SRSTU} & R_1, R_2 & \text{[RRE]} \\
\hline
\text{‘B9BE’} & /\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\\\!\\\end{array}
\]

The second operand is searched until a specified two-byte character is found, the end of the second operand is reached, or a CPU-determined number of two-byte characters have been searched, whichever occurs first. The CPU-determined number is at least 256 two-byte characters. The result is indicated in the condition code.

The location of the leftmost two-byte character of the second operand is designated by the contents of general register \( R_2 \). The location of the first two-byte character after the second operand is designated by the contents of general register \( R_1 \).

The handling of the addresses in general registers \( R_1 \) and \( R_2 \) is dependent on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers \( R_1 \) and \( R_2 \) constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

In the access-register mode, the address space containing the second operand is specified only by means of access register \( R_2 \). The contents of access register \( R_1 \) are ignored.

The two-byte character for which the search occurs is specified in bit positions 48-63 of general register 0. Bit positions 32-47 of general register 0 are reserved for possible future extensions and must contain all zeros; otherwise, a specification exception is recognized.

The operation proceeds left to right in steps of two bytes and ends as soon as the specified two-byte character has been found in the second operand, the address of the next second-operand two-byte character to be examined is equal or greater than the address in general register \( R_1 \), or a...
CPU-determined number of second-operand two-byte characters have been examined, whichever occurs first. The CPU-determined number is at least 256. When the specified two-byte character is found, condition code 1 is set. When the address of the second-operand two-byte character to be examined is equal or greater than the address in general register R, condition code 2 is set. When a CPU-determined number of second-operand two-byte characters have been examined, condition code 3 is set. When the CPU-determined number of second-operand two-byte characters have been examined and the address of the next second-operand two-byte character is in general register R, it is unpredictable whether condition code 2 or 3 is set.

When condition code 1 is set, the address of the specified two-byte character found in the second operand is placed in general register R, and the contents of general register R remain unchanged. When condition code 3 is set, the address of the next two-byte character to be processed in the second operand is placed in general register R, and the contents of general register R remain unchanged. When condition code 2 is set, the contents of general registers R and R remain unchanged. Whenever an address is placed in a general register, bits 32-63 of the register, in the 24-bit addressing mode, or bit 32, in the 31-bit addressing mode, are set to zeros. Bits 0-31 of the R and R registers always remain unchanged in the 24-bit or 31-bit mode.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

Access exceptions for the second operand are recognized only for that portion of the operand that is necessarily examined.

The storage-operand-consistency rules are the same as for the COMPARE LOGICAL LONG instruction.

\[\text{RESULTING CONDITION CODE:}\]

0 --

1 Specified two-byte character found; general register R updated with address of character; general register R unchanged
2 Specified two-byte character not found in entire second operand; general registers R and R unchanged
3 CPU-determined number of two-byte characters searched; general register R unchanged; general register R updated with address of next two-byte character

\textbf{Program Exceptions:}

- Access (fetch, operand 2)
- Operation (if the extended-translation facility 3 is not installed)
- Specification

\textbf{Programming Notes:}

1. When condition code 3 is set, the program can simply branch back to the instruction to continue the search. The program need not determine the number of two-byte characters that were searched.
2. When the address in general register R equals the address in general register R, condition code 2 is set immediately, and access exceptions are not recognized. When the address in general register R is less than the address in general register R, condition code 2 can be set only if the operand wraps around from the top of storage to location 0.
3. R or R may be zero, in which case general register 0 is treated as containing an address and also the specified character.

\textbf{SET ACCESS}

\[\text{SAR } R_1,R_2 [\text{RRE}]\]

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
'B24E' & \multicolumn{3}{|c|}{\text{"/ / / / / /"}} \\
\hline
0 & 16 & 24 & 28 & 31 \\
\hline
\end{tabular}
\end{center}

The contents of bit positions 32-63 of general register R are placed in access register R.

\textbf{Condition Code:} The code remains unchanged.

\textbf{Program Exceptions:} None.
SET ADDRESSING MODE

The addressing mode is set by setting the extended-addressing-mode bit, bit 31 of the current PSW, and the basic-addressing-mode bit, bit 32 of the current PSW, as follows:

<table>
<thead>
<tr>
<th>Instruction PSW Bit</th>
<th>PSW Bit</th>
<th>Resulting Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAM24</td>
<td>0</td>
<td>24-bit</td>
</tr>
<tr>
<td>SAM31</td>
<td>0</td>
<td>31-bit</td>
</tr>
<tr>
<td>SAM64</td>
<td>1</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

The instruction address in the PSW is updated under the control of the new addressing mode, as follows. The value 2 (the instruction length) is added to the contents of bit positions 64-127 of the PSW, or the value 4 is added if the instruction is the target of EXECUTE. In either case, a carry out of bit position 0 is ignored. Then bits 64-103 of the PSW are set to zeros if the new addressing mode is the 24-bit mode, or bits 64-96 are set to zeros if the new addressing mode is the 31-bit mode.

The instruction is completed only if the new addressing mode and the unupdated instruction address in the PSW are a valid combination. When the new addressing mode is to be the 24-bit mode, bits 64-103 of the unupdated PSW must be all zeros, or, when the new addressing mode is to be the 31-bit mode, bits 64-96 of the unupdated PSW must be all zeros; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:
- Specification (SAM24 and SAM31 only)
- Trace

Programming Note: Checking the unupdated instruction address prevents completion in two major cases: the instruction is located at address \(2^{24}\) or above and the new addressing mode is to be the 24-bit mode, or the instruction is located at address \(2^{31}\) or above and the new addressing mode is to be the 24-bit or 31-bit mode. In these cases, if the instruction were completed, the updating of the instruction address under the control of the new addressing mode would cause one or more leftmost bits of the address to be set to zeros, which would cause the next instruction to be fetched from other than the next sequential location. This action is sometimes called a “wild branch.” A wild branch still can occur if the instruction is located at \(2^{24} - 2\) or \(2^{31} - 2\), or at \(2^{24} - 4\) or \(2^{31} - 4\) if EXECUTE is used.

SET PROGRAM MASK

The first operand is used to set the condition code and the program mask of the current PSW.

Bits 34 and 35 of general register \(R_1\) replace the condition code, and bits 36-39 replace the program mask. Bits 0-33 and 40-63 of general register \(R_1\) are ignored.

Condition Code: The code is set as specified by bits 34 and 35 of general register \(R_1\).

Program Exceptions: None.

Programming Notes:
1. Bits 34-39 of the general register may have been loaded from the PSW by execution of BRANCH AND LINK in the 24-bit addressing mode or by execution of INSERT PROGRAM MASK in either the 24-bit or 31-bit addressing mode.

2. SET PROGRAM MASK permits setting of the condition code and the mask bits in either the problem state or the supervisor state.
3. The program should take into consideration that the setting of the program mask can have a significant effect on subsequent execution of the program. Not only do the four mask bits control whether the corresponding interruptions occur, but the exponent-underflow and significance masks also determine the result which is obtained.

### SHIFT LEFT DOUBLE

**SLDA**  
$R_1, D_2(B_2)$  
$[RS]$  

<table>
<thead>
<tr>
<th>'8F'</th>
<th>$R_1$</th>
<th>$B_2$</th>
<th>$D_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

The 63-bit numeric part of the signed first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register $R_1$ followed on the right by bits 32-63 of general register $R_1 + 1$.

The $R_1$ field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

The first operand is treated as a 64-bit signed binary integer. The sign bit of the first operand, bit 32 of the even-numbered register, remains unchanged. Bit position 32 of the odd-numbered register contains a numeric bit, which participates in the shift in the same manner as the other numeric bits. Zeros are supplied to the vacated bit positions on the right. Bits 0-31 of general registers $R_1$ and $R_1 + 1$ remain unchanged.

If one or more bits unlike the sign bit are shifted out of bit position 33 of the even-numbered register, an overflow occurs, and condition code 3 is set. If the fixed-point-overflow mask bit is one, a program interruption for fixed-point overflow occurs.

**Resulting Condition Code:**

0. Result zero; no overflow  
1. Result less than zero; no overflow  
2. Result greater than zero; no overflow  
3. Overflow

**Program Exceptions:**
- Fixed-point overflow
- Specification

**Programming Notes:**
1. An example of the use of the SHIFT LEFT DOUBLE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. The eight shift instructions that are in both ESA/390 and z/Architecture provide the following three pairs of alternatives for 32 bits in one general register or, for double, in each of two general registers: left or right, single or double, and signed or logical. The four additional shift instructions in z/Architecture provide left or right, signed or logical shifts of 64 bits in one general register. The signed shifts differ from the logical shifts in that, in the signed shifts, overflow is recognized, the condition code is set, and the leftmost bit participates as a sign.

3. A zero shift amount in the two signed double-shift operations provides a double-length sign and magnitude test.

4. The base register participating in the generation of the second-operand address permits indirect specification of the shift amount by means of placement of the shift amount in the base register. A zero in the $B_2$ field indicates the absence of indirect shift specification.

### SHIFT LEFT DOUBLE LOGICAL

**SLDL**  
$R_1, D_2(B_2)$  
$[RS]$  

<table>
<thead>
<tr>
<th>'8D'</th>
<th>$R_1$</th>
<th>$B_2$</th>
<th>$D_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

The 64-bit first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register $R_1$ followed on the right by bits 32-63 of general register $R_1 + 1$. 

---

7-176  
z/Architecture Principles of Operation
The R_i field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

All 64 bits of the first operand participate in the shift. Bits shifted out of bit position 32 of the even-numbered register are not inspected and are lost. Zeros are supplied to the vacated bit positions on the right. Bits 0-31 of general registers R_i and R_i + 1 remain unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Specification

### SHIFT LEFT SINGLE

**SLA**  
\[ R_1, D_2(B_2) \quad [RS] \]

<table>
<thead>
<tr>
<th>'8B'</th>
<th>R_1</th>
<th>// //</th>
<th>B_2</th>
<th>D_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

**SLAG**  
\[ R_1, R_3, D_2(B_2) \quad [RSY] \]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R_1</th>
<th>R_3</th>
<th>B_2</th>
<th>DL_2</th>
<th>DH_2</th>
<th>'OB'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

For SHIFT LEFT SINGLE (SLA), the 31-bit numeric part of the signed first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R_1 remain unchanged.

For SHIFT LEFT SINGLE (SLAG), the 63-bit numeric part of the signed third operand is shifted left the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is placed at the first-operand location. The third operand remains unchanged in general register R_3.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SLA, the first operand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R_1. The sign of the first operand remains unchanged. All 31 numeric bits of the operand participate in the left shift.

For SLAG, the first and third operands are treated as 64-bit signed binary integers in bit positions 0-63 of general registers R_1 and R_3, respectively. The sign of the first operand is set equal to the sign of the third operand. All 63 numeric bits of the third operand participate in the left shift.

For SLA or SLAG, zeros are supplied to the vacated bit positions on the right.

If one or more bits unlike the sign bit are shifted out of bit position 33, for SLA, or 1, for SLAG, an overflow occurs, and condition code 3 is set. If the fixed-point-overflow mask bit is one, a program interruption for fixed-point overflow occurs.

**Resulting Condition Code:**
- 0 Result zero; no overflow
- 1 Result less than zero; no overflow
- 2 Result greater than zero; no overflow
- 3 Overflow

**Program Exceptions:**
- Fixed-point overflow

**Programming Notes:**

1. An example of the use of the SHIFT LEFT SINGLE instruction is given in **Appendix A. “Number Representation and Instruction-Use Examples.”**

2. For SHIFT LEFT SINGLE (SLA), for numbers with a value greater than or equal to \( -2^{10} \) and less than \( 2^{10} \), a left shift of one bit position is equivalent to multiplying the number by 2. For SHIFT LEFT SINGLE (SLAG), the comparable values are \( -2^{62} \) and \( 2^{62} \).

3. For SHIFT LEFT SINGLE (SLA), shift amounts from 31 to 63 cause the entire numeric part to be shifted out of the register, leaving a result of the maximum negative number or zero, depending on whether or not the initial contents were negative. For SHIFT...
LEFT SINGLE (SLAG), a shift amount of 63 causes the same effect.

**SHIFT LEFT SINGLE LOGICAL**

SLL  \( R_{1, D2} (B_2) \)  [RS]

<table>
<thead>
<tr>
<th>'89'</th>
<th>'R'</th>
<th>'/'</th>
<th>'B'</th>
<th>'D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

SLLG \( R_{1, R3, D2} (B_2) \)  [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>'R'</th>
<th>'R'</th>
<th>'B'</th>
<th>'DL'</th>
<th>'DH'</th>
<th>'0D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

For **SHIFT LEFT SINGLE LOGICAL** (SLL), the 32-bit first operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register \( R_1 \) remain unchanged.

For **SHIFT LEFT SINGLE LOGICAL** (SLLG), the 64-bit third operand is shifted left the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register \( R_3 \).

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SLL, the first operand is in bit positions 32-63 of general register \( R_1 \). All 32 bits of the operand participate in the left shift.

For SLLG, the first and third operands are in bit positions 0-63 of general registers \( R_1 \) and \( R_3 \), respectively. All 64 bits of the third operand participate in the left shift.

For SLL or SLLG, zeros are supplied to the vacated bit positions on the right.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**SHIFT RIGHT DOUBLE**

SRDA \( R_{1, D2} (B_2) \)  [RS]

<table>
<thead>
<tr>
<th>'8E'</th>
<th>'R'</th>
<th>'/'</th>
<th>'B'</th>
<th>'D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The 63-bit numeric part of the signed first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register \( R_1 \) followed on the right by bits 32-63 of general register \( R_1 + 1 \).

The \( R_1 \) field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

The first operand is treated as a 64-bit signed binary integer. The sign bit of the first operand, bit 32 of the even-numbered register, remains unchanged. Bit position 32 of the odd-numbered register contains a numeric bit, which participates in the shift in the same manner as the other numeric bits. Bits shifted out of bit position 63 of the odd-numbered register are not inspected and are lost. Bits equal to the sign are supplied to the vacated bit positions on the left. Bits 0-31 of general registers \( R_1 \) and \( R_1 + 1 \) remain unchanged.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>0</th>
<th>Result zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Result less than zero</td>
</tr>
<tr>
<td>2</td>
<td>Result greater than zero</td>
</tr>
<tr>
<td>3</td>
<td>--</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Specification
SHIFT RIGHT DOUBLE LOGICAL

SRDL  R₁, D₂(B₂)  [RS]

<table>
<thead>
<tr>
<th>'8C'</th>
<th>R₁</th>
<th>/////</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The 64-bit first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The first operand consists of bits 32-63 of general register R₁ followed on the right by bits 32-63 of general register R₁ + 1.

The R₁ field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

All 64 bits of the first operand participate in the shift. Bits shifted out of bit position 63 of the odd-numbered register are not inspected and are lost. Zeros are supplied to the vacated bit positions on the left. Bits 0-31 of general registers R₁ and R₁ + 1 remain unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Specification

SHIFT RIGHT SINGLE

SRA  R₁, D₂(B₂)  [RS]

<table>
<thead>
<tr>
<th>'8A'</th>
<th>R₁</th>
<th>/////</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

SRAG  R₁, R₃, D₂(B₂)  [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R₁</th>
<th>R₃</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'0A'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

For SHIFT RIGHT SINGLE (SRA), The 31-bit numeric part of the signed first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-32 of general register R₁ remain unchanged.

For SHIFT RIGHT SINGLE (SRAG), the 63-bit numeric part of the signed third operand is shifted right the number of bits specified by the second-operand address, and the result, with the sign bit of the third operand appended on its left, is placed at the first-operand location. The third operand remains unchanged in general register R₃.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SRA, The first operand is treated as a 32-bit signed binary integer in bit positions 32-63 of general register R₁. The sign of the first operand remains unchanged. All 31 numeric bits of the operand participate in the right shift.

For SRAG, the first and third operands are treated as 64-bit signed binary integers in bit positions 0-63 of general registers R₁ and R₁, respectively. The sign of the first operand is set equal to the sign of the third operand. All 63 numeric bits of the third operand participate in the right shift.

For SRA or SRAG, bits shifted out of bit position 63 are not inspected and are lost. Bits equal to the sign are supplied to the vacated bit positions on the left.

**Resulting Condition Code:**
- 0 Result zero
- 1 Result less than zero
- 2 Result greater than zero
- 3 --

**Program Exceptions:** None.

**Programming Notes:**
- A right shift of one bit position is equivalent to division by 2 with rounding downward. When an even number is shifted right one position, the result is equivalent to dividing the number by 2. When an odd number is shifted right one position, the result is equivalent to dividing the next lower number by 2. For
example, +5 shifted right by one bit position yields +2, whereas -5 yields -3.

2. For SHIFT RIGHT SINGLE (SRA), shift amounts from 31 to 63 cause the entire numeric part to be shifted out of the register, leaving a result of -1 or zero, depending on whether or not the initial contents were negative. For SHIFT RIGHT SINGLE (SRAG), a shift amount of 63 causes the same effect.

**SHIFT RIGHT SINGLE LOGICAL**

<table>
<thead>
<tr>
<th>SRL</th>
<th>R₁, D₂ (B₂)</th>
<th>[RS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'88'</td>
<td>R₁, B₂</td>
<td>D₂</td>
</tr>
</tbody>
</table>

| 0  | 8  | 12 | 16 | 20 | 31 |

<table>
<thead>
<tr>
<th>SRLG</th>
<th>R₁, R₃, D₂ (B₂)</th>
<th>[RSY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EB'</td>
<td>R₁, R₃, B₂</td>
<td>D₂, D₃</td>
</tr>
</tbody>
</table>

| 0  | 8  | 12 | 16 | 20 | 32 | 40 | 47 |

For SHIFT RIGHT SINGLE LOGICAL (SRL), the 32-bit first operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. Bits 0-31 of general register R₁ remain unchanged.

For SHIFT RIGHT SINGLE LOGICAL (SRLG), the 64-bit third operand is shifted right the number of bits specified by the second-operand address, and the result is placed at the first-operand location. The third operand remains unchanged in general register R₃.

The second-operand address is not used to address data; its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

For SRL, the first operand is in bit positions 32-63 of general register R₁. All 32 bits of the operand participate in the right shift.

For SRLG, the first and third operands are in bit positions 0-63 of general registers R₁ and R₃, respectively. All 64 bits of the third operand participate in the right shift.

For SRL or SRLG, bits shifted out of bit position 63 are not inspected and are lost. Zeros are supplied to the vacated bit positions on the left.

**Condition Code:** The code remains unchanged.

**Program Exceptions:** None.

**STORE**

<table>
<thead>
<tr>
<th>ST</th>
<th>R₁, D₂ (X₂, B₂)</th>
<th>[RX]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'50'</td>
<td>R₁, B₂</td>
<td>D₂</td>
</tr>
</tbody>
</table>

| 0  | 8  | 12 | 16 | 20 | 31 |

<table>
<thead>
<tr>
<th>STY</th>
<th>R₁, D₂ (X₂, B₂)</th>
<th>[RXY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E3'</td>
<td>R₁, X₂, B₂</td>
<td>D₂, D₃</td>
</tr>
</tbody>
</table>

| 0  | 8  | 12 | 16 | 20 | 32 | 40 | 47 |

<table>
<thead>
<tr>
<th>STG</th>
<th>R₁, D₂ (X₂, B₂)</th>
<th>[RXY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E3'</td>
<td>R₁, X₂, B₂</td>
<td>D₂, D₃</td>
</tr>
</tbody>
</table>

| 0  | 8  | 12 | 16 | 20 | 32 | 40 | 47 |

The first operand is placed unchanged at the second-operand location.

For STORE (ST, STY), the operands are 32 bits, and, for STORE (STG), the operands are 64 bits.

The displacement for ST is treated as a 12-bit unsigned binary integer. The displacement for STY and STG is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (store, operand 2)
- Operation (STY, if the long-displacement facility is not installed)
STORE ACCESS MULTIPLE

STAM  R₁, R₃, D₂ (B₂)  [RS]

<table>
<thead>
<tr>
<th>'9B'</th>
<th>R₁</th>
<th>R₃</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

STAMY  R₁, R₃, D₂ (B₂)  [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R₁</th>
<th>R₃</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'9B'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

The contents of the set of access registers starting with access register R₁ and ending with access register R₃ are stored at the locations designated by the second-operand address.

The storage area where the contents of the access registers are placed starts at the location designated by the second-operand address and continues through as many storage words as the number of access registers specified. The contents of the access registers are stored in ascending order of their register numbers, starting with access register R₁ and continuing up to and including access register R₃, with access register 0 following access register 15. The contents of the access registers remain unchanged.

The displacement for STAM is treated as a 12-bit unsigned binary integer. The displacement for STAMY is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation (STAMY, if the long-displacement facility is not installed)

STORE CHARACTER

STC  R₁, D₂ (X₂, B₂)  [RX]

<table>
<thead>
<tr>
<th>'42'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

STCY  R₁, D₂ (X₂, B₂)  [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'72'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Bits 56-63 of general register R₁ are placed unchanged at the second-operand location. The second operand is one byte in length.

The displacement for STC is treated as a 12-bit unsigned binary integer. The displacement for STCY is treated as a 20-bit signed binary integer.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (store, operand 2)
- Operation (STCY, if the long-displacement facility is not installed)

STORE CHARACTERS UNDER MASK

STCM  R₁, M₃, D₂ (B₂)  [RS]

<table>
<thead>
<tr>
<th>'BE'</th>
<th>R₁</th>
<th>M₃</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

STCMY  R₁, M₃, D₂ (B₂)  [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R₁</th>
<th>M₃</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'2D'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

STCMH  R₁, M₃, D₂ (B₂)  [RSY]

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R₁</th>
<th>M₃</th>
<th>B₂</th>
<th>DL₂</th>
<th>DH₂</th>
<th>'2C'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Bytes selected from general register R₁ under control of a mask are placed at contiguous byte
locations beginning at the second-operand address.

The contents of the M3 field are used as a mask. These four bits, left to right, correspond one for one with four bytes, left to right, of general register R1. For STORE CHARACTERS UNDER MASK (STCM, STCMY), the four bytes to which the mask bits correspond are in bit positions 32-63 of general register R1. For STORE CHARACTERS UNDER MASK (STCMH), the four bytes are in the high-order half, bit positions 0-31, of the register. The bytes corresponding to ones in the mask are placed in the same order at successive and contiguous storage locations beginning at the second-operand address. When the mask is not zero, the length of the second operand is equal to the number of ones in the mask. The contents of the general register remain unchanged.

When the mask is not zero, exceptions associated with storage-operand accesses are recognized only for the number of bytes specified by the mask.

When the mask is zero, the single byte designated by the second-operand address remains unchanged; however, on some models, the contents may be fetched and subsequently stored back unchanged at the same storage location. This update appears to be an interlocked-update reference as observed by other CPUs.

The displacement for STCM is treated as a 12-bit unsigned binary integer. The displacement for STCMY and STCMH is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Operation (STCMY, if the long-displacement facility is not installed)

**Programming Notes:**
1. An example of the use of the STORE CHARACTERS UNDER MASK instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. STORE CHARACTERS UNDER MASK (STCM, STCMY), with a mask of 0111 may be used to store a three-byte address, for example, in modifying the address in a CCW.
3. STORE CHARACTERS UNDER MASK (STCM, STCMY) with a mask of 1111, 0011, or 0001 performs the same function as STORE (ST), STORE HALFWORD, or STORE CHARACTER, respectively. However, on most models, the performance of STORE CHARACTERS UNDER MASK is slower.
4. Using STORE CHARACTERS UNDER MASK with a zero mask should be avoided since this instruction, depending on the model, may perform a fetch and store of the single byte designated by the second-operand address. This reference is not interlocked against accesses by channel programs. In addition, it may cause any of the following to occur for the byte designated by the second-operand address: a PER storage-alteration event may be recognized; access exceptions may be recognized; and, provided no access exceptions exist, the change bit may be set to one. Because the contents of storage remain unchanged, the change bit may or may not be one when a PER storage-alteration event is recognized.

### STORE CLOCK

**STCK** D2(B2) [S]

<table>
<thead>
<tr>
<th>'B205'</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The current value of bits 0-63 of the TOD clock is stored in the eight-byte field designated by the second-operand address, provided the clock is in the set, stopped, or not-set state.

When the clock is stopped, zeros are stored in positions to the right of the rightmost bit position that is incremented when the clock is running. When the value of a running clock is stored, nonzero values may be stored in positions to the right of the rightmost incremented bit; this is to ensure that a unique value is stored.

Zeros are stored for the rightmost bit positions that are not provided by the clock.
Zeros are stored at the operand location when the clock is in the error state or the not-operational state.

The quality of the clock value stored by the instruction is indicated by the resultant condition-code setting.

A serialization function is performed before the value of the clock is fetched and again after the value is placed in storage.

**Resulting Condition Code:**

0 Clock in set state  
1 Clock in not-set state  
2 Clock in error state  
3 Clock in stopped state or not-operational state

**Program Exceptions:**

- Access (store, operand 2)

**Programming Notes:**

1. Bit position 31 of the clock is incremented every 1.048576 seconds; hence, for timing applications involving human responses, the leftmost clock word may provide sufficient resolution.

2. Condition code 0 normally indicates that the clock has been set by the control program. Accordingly, the value may be used in elapsed-time measurements and as a valid time-of-day and calendar indication. Condition code 1 indicates that the clock value is the elapsed time since the power for the clock was turned on. In this case, the value may be used in elapsed-time measurements but is not a valid time-of-day indication. Condition codes 2 and 3 mean that the value provided by STORE CLOCK cannot be used for time measurement or indication.

3. Condition code 3 indicates that the clock is in either the stopped state or the not-operational state. These two states can normally be distinguished because an all-zero value is stored when the clock is in the not-operational state.

4. If a problem program written for z/Architecture is to be executed also on a system in the System/370 mode, then the program should take into account that, in the System/370 mode, the value stored when the condition code is 2 is not necessarily zero.

---

**STORE CLOCK EXTENDED**

\[
\text{STCKE } D_z(B_z) \quad [S]
\]

<table>
<thead>
<tr>
<th>'8278'</th>
<th>B_z</th>
<th>D_z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The current value of bits 0-103 of the TOD clock is stored in byte positions 1-13 of the sixteen-byte field designated by the second-operand address, provided the clock is in the set, stopped, or not-set state. Zeros are stored in byte position 0. The TOD programmable field, bits 16-31 of the TOD programmable register, is stored in byte positions 14 and 15.

The operand just described has the following format:

<table>
<thead>
<tr>
<th>Zeros</th>
<th>TOD Clock</th>
<th>Programmable Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>112 127</td>
</tr>
</tbody>
</table>

When the clock is stopped, zeros are stored in the clock value in positions to the right of the rightmost bit position that is incremented when the clock is running. The programmable field still is stored.

When the value of a running clock is stored, the value in bit positions 64-103 of the clock (bit positions 72-111 of the storage operand) is always nonzero; this ensures that values stored by STORE CLOCK EXTENDED are unique when compared with values stored by STORE CLOCK and extended with zeros.

Zeros are stored at the operand location when the clock is in the error state or the not-operational state.

The quality of the clock value stored by the instruction is indicated by the resultant condition-code setting.

A serialization function is performed before the value of the clock is fetched and again after the value is placed in storage.

**Resulting Condition Code:**

0 Clock in set state  
1 Clock in not-set state
2 Clock in error state
3 Clock in stopped state or not-operational state

**Program Exceptions:**

- Access (store, operand 2)

**Programming Notes:**

1. Condition code 0 normally indicates that the clock has been set by the control program. Accordingly, the value may be used in elapsed-time measurements and as a valid time-of-day and calendar indication. Condition code 1 indicates that the clock value is the elapsed time since the power for the clock was turned on. In this case, the value may be used in elapsed-time measurements but is not a valid time-of-day indication. Condition codes 2 and 3 mean that the value provided by STORE CLOCK EXTENDED cannot be used for time measurement or indication.

2. Programming notes beginning on page 4-43 show hex values related to the value of the TOD clock as it is stored by the STORE CLOCK instruction. Notes 3-5, below, are repetitions of those notes except with the text and hex values adjusted so they apply to bits 0-71 of the value stored by STORE CLOCK EXTENDED.

3. The following chart shows the time interval between instants at which various bits of the TOD-clock value stored by STORE CLOCK EXTENDED are stepped. This time value may also be considered as the weighted time value that the bit, when one, represents. The bit numbers are those of the STORE CLOCK EXTENDED operand.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Stepping Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Days</td>
</tr>
<tr>
<td>59</td>
<td>0.000</td>
</tr>
<tr>
<td>55</td>
<td>0.000</td>
</tr>
<tr>
<td>51</td>
<td>0.000</td>
</tr>
<tr>
<td>47</td>
<td>0.004</td>
</tr>
<tr>
<td>43</td>
<td>0.065</td>
</tr>
<tr>
<td>39</td>
<td>1.048</td>
</tr>
<tr>
<td>35</td>
<td>16.777</td>
</tr>
<tr>
<td>31</td>
<td>28.435</td>
</tr>
<tr>
<td>27</td>
<td>34.967</td>
</tr>
<tr>
<td>23</td>
<td>19.476</td>
</tr>
<tr>
<td>19</td>
<td>12.017</td>
</tr>
<tr>
<td>15</td>
<td>25.116</td>
</tr>
<tr>
<td>11</td>
<td>36.710</td>
</tr>
</tbody>
</table>

4. The following chart shows the setting of bits 0-63 of the STORE CLOCK EXTENDED operand for 00:00:00 (0 am), UTC time, for several dates: January 1, 1900, January 1, 1972, and for that instant in time just after each of the 22 leap seconds that have occurred through January, 1999. Each of these leap seconds was inserted in the UTC time scale beginning at 23:59:60 UTC of the day previous to the one listed and ending at 00:00:00 UTC of the day listed.
The stepping value of TOD-clock bit position 63, if implemented, is $2^{-12}$ microseconds, or approximately 244 picoseconds. This value is called a clock unit.

The following chart shows various time intervals in clock units expressed in hexadecimal notation. The chart shows the values stored in bit positions 0-71 of the STORE CLOCK EXTENDED operand. Bit 71 of the operand represents a clock unit.

<table>
<thead>
<tr>
<th>Interval</th>
<th>Clock Units (Hex)</th>
<th>Bits 0-71</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 microsecond</td>
<td>0010 00</td>
<td>00</td>
</tr>
<tr>
<td>1 millisecond</td>
<td>00F4 2400 00</td>
<td>00</td>
</tr>
<tr>
<td>1 second</td>
<td>3938 7000 00</td>
<td>00</td>
</tr>
<tr>
<td>1 minute</td>
<td>000D 693A 4000 00</td>
<td>00</td>
</tr>
<tr>
<td>1 hour</td>
<td>0141 D076 0000</td>
<td>00</td>
</tr>
<tr>
<td>365 days</td>
<td>0001 CAEB C13E 0000</td>
<td>00</td>
</tr>
<tr>
<td>366 days</td>
<td>0001 CC2A 9EB4 0000</td>
<td>00</td>
</tr>
<tr>
<td>1,461 days*</td>
<td>0007 2CE4 E26E 0000</td>
<td>00</td>
</tr>
</tbody>
</table>

* Number of days in four years, including a leap year. Note that the year 1900 was not a leap year. Thus, the four-year span starting in 1900 has only 1,460 days.

Bits 48-63 of general register R1 are placed unchanged at the second-operand location. The second operand is two bytes in length.

The displacement for STH is treated as a 12-bit unsigned binary integer. The displacement for STHY is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Operation (STHY if the long-displacement facility is not installed)

### STORE HALFWORD

<table>
<thead>
<tr>
<th>STH</th>
<th>R1, D2 (X2, B2) [RX]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'40'</td>
<td>R1 X2 B2 D2</td>
</tr>
<tr>
<td>0</td>
<td>8 12 16 20 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STHY</th>
<th>R1, D2 (X2, B2) [RX]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E3'</td>
<td>R1 X2 B2 DL2 DH2 70'</td>
</tr>
<tr>
<td>0</td>
<td>8 12 16 20 32 40 47</td>
</tr>
</tbody>
</table>

The contents of bit positions of the set of general registers starting with general register R1 and ending with general register R3 are placed in the storage area beginning at the location designated...
by the second-operand address and continuing through as many locations as needed.

For STORE MULTIPLE (STM, STMY), the contents of bit positions 32-63 of the general registers are stored in successive four-byte fields beginning at the second-operand address. For STORE MULTIPLE (STMG), the contents of bit positions 0-63 of the general registers are stored in successive eight-byte fields beginning at the second-operand address.

The general registers are stored in the ascending order of their register numbers, starting with general register R1 and continuing up to and including general register R3, with general register 0 following general register 15.

The displacement for STM is treated as a 12-bit unsigned binary integer. The displacement for STMY and STMG is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Operation (STMY, if the long-displacement facility is not installed)

**Programming Note:** An example of the use of the STORE MULTIPLE instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

### STORE MULTIPLE HIGH

**STMH**  
R1,R2,D2(X2,B2)  [RSY]

The contents of the high-order halves, bit positions 0-31, of the set of general registers starting with general register R1 and ending with general register R3 are placed in the storage area beginning at the location designated by the second-operand address and continuing through as many locations as needed, that is, the contents of bit positions 0-31 are stored in successive four-byte fields beginning at the second-operand address. Bits 32-63 of the registers are ignored.

The general registers are stored in the ascending order of their register numbers, starting with general register R1 and continuing up to and including general register R3, with general register 0 following general register 15.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Specification

**Programming Notes:**
1. The STORE MULTIPLE (STM or STMG) instruction does not necessarily provide quadword-concurrent access.
2. The performance of STORE PAIR TO QUADWORD on some models may be significantly slower than that of STORE MULTIPLE.
(STMG). Unless quadword consistency is required, STMG should be used instead of STPQ.

**STORE REVERSED**

```plaintext
STRH R1,Dz(X2,B2) [RXY]

| 'E3' | R1 | X2 | B2 | DLz | DHz | '3F'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

STRV R1,Dz(X2,B2) [RXY]

| 'E3' | R1 | X2 | B2 | DLz | DHz | '3E'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

STRVG R1,Dz(X2,B2) [RXY]

| 'E3' | R1 | X2 | B2 | DLz | DHz | '2F'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>
```

The first operand is placed at the second-operand location with the left-to-right sequence of the bytes reversed.

For STORE REVERSED (STRH), the first operand is two bytes in bit positions 48-63 of general register \( R_1 \). For STORE REVERSED (STRV), the first operand is four bytes in bit positions 32-63 of general register \( R_1 \). For STORE REVERSED (STRVG), the first operand is eight bytes in bit positions 0-63 of general register \( R_1 \).

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)

**Programming Notes:**
1. The instruction can be used to convert two, four, or eight bytes from a "little-endian" format to a "big-endian" format, or vice versa. In the big-endian format, the bytes in a left-to-right sequence are in the order most significant to least significant. In the little-endian format, the bytes are in the order least significant to most significant. For example, the bytes ABCD in the big-endian format are DCBA in the little-endian format.

2. The storage-operand references of STORE REVERSED may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

**SUBTRACT**

```plaintext
SR R1,R2 [RR]

<table>
<thead>
<tr>
<th>'1B'</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

SGR R1,R2 [RRE]

<table>
<thead>
<tr>
<th>'B909'</th>
<th>/////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

SGFR R1,R2 [RRE]

<table>
<thead>
<tr>
<th>'B919'</th>
<th>/////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

S R1,Dz(X2,B2) [RX]

<table>
<thead>
<tr>
<th>'5B'</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>Dz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

SY R1,Dz(X2,B2) [RXY]

| 'E3' | R1 | X2 | B2 | DLz | DHz | '5B'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

SG R1,Dz(X2,B2) [RXY]

| 'E3' | R1 | X2 | B2 | DLz | DHz | '09'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

SGF R1,Dz(X2,B2) [RXY]

| 'E3' | R1 | X2 | B2 | DLz | DHz | '19'
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>
```

The second operand is subtracted from the first operand, and the difference is placed at the first-operand location. For SUBTRACT (SR, S, SY), the operands and the difference are treated as 32-bit signed binary integers. For SUBTRACT
(SGR, SG), they are treated as 64-bit signed binary integers. For SUBTRACT (SGFR, SGF), the second operand is treated as a 32-bit signed binary integer, and the first operand and the difference are treated as 64-bit signed binary integers.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

The displacement for S is treated as a 12-bit unsigned binary integer. The displacement for SY, SG, and SGF is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0  Result zero; no overflow
1  Result less than zero; no overflow
2  Result greater than zero; no overflow
3  Overflow

Program Exceptions:

- Access (fetch, operand 2 of S, SY, SG, and SGF only)
- Fixed-point overflow
- Operation (SY, if the long-displacement facility is not installed)

Programming Notes:

1. For SR and SGR, when R1 and R2 designate the same register, subtracting is equivalent to clearing the register.
2. Subtracting a maximum negative number from itself gives a zero result and no overflow.

SUBTRACT HALFWORD

SH  R1,D2(X2,B2)  [RX]

\[\begin{array}{ccccc}
\text{'4B'} & R1 & X2 & B2 & D2 \\
0 & 8 & 12 & 16 & 20 & 31 \end{array}\]

SHY  R1,D2(X2,B2)  [RXY]

\[\begin{array}{ccccc}
\text{'E3'} & R1 & X2 & B2 & DL2 & DH2 & '7B' \\
0 & 8 & 12 & 16 & 20 & 32 & 40 & 47 \end{array}\]

The second operand is subtracted from the first operand, and the difference is placed at the first-operand location. The second operand is two bytes in length and is treated as a 16-bit signed binary integer. The first operand and the difference are treated as 32-bit signed binary integers.

When there is an overflow, the result is obtained by allowing any carry into the sign-bit position and ignoring any carry out of the sign-bit position, and condition code 3 is set. If the fixed-point-overflow mask is one, a program interruption for fixed-point overflow occurs.

The displacement for SH is treated as a 12-bit unsigned binary integer. The displacement for SHY is treated as a 20-bit signed binary integer.

Resulting Condition Code:

0  Result zero; no overflow
1  Result less than zero; no overflow
2  Result greater than zero; no overflow
3  Overflow

Program Exceptions:

- Access (fetch, operand 2)
- Fixed-point overflow
- Operation (SHY, if the long-displacement facility is not installed)

Programming Note: The function of a SUBTRACT HALFWORD IMMEDIATE instruction, which is an instruction not provided, can be obtained by using an ADD HALFWORD IMMEDIATE instruction with a negative I2 field.

SUBTRACT LOGICAL

SLR  R1,R2  [RR]

\[\begin{array}{ccc}
'1F' & R1 & R2 \\
0 & 8 & 12 & 15 \end{array}\]

SLGR  R1,R2  [RRE]

\[\begin{array}{cccc}
'B90B' & R1 & R2 & //\\n0 & 16 & 24 & 28 & 31 \end{array}\]
The second operand is subtracted from the first operand, and the difference is placed at the first-operand location. For SUBTRACT LOGICAL (SLR, SL, SLY), the operands and the difference are treated as 32-bit unsigned binary integers. For SUBTRACT LOGICAL (SLGR, SLG), they are treated as 64-bit unsigned binary integers. For SUBTRACT LOGICAL (SLGFR, SLGF) the second operand is treated as a 32-bit unsigned binary integer, and the first operand and the difference are treated as 64-bit unsigned binary integers.

The displacement for SL is treated as a 12-bit unsigned binary integer. The displacement for SLY, SLG, and SLGF is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

0  --
1  Result not zero; borrow
2  Result zero; no borrow
3  Result not zero; no borrow

**Program Exceptions:**

- Access (fetch, operand 2 of SL, SLY, SLG, and SLGF only)
- Operation (SLY, if the long-displacement facility is not installed)

**Programming Notes:**

1. Logical subtraction is performed by adding the one’s complement of the second operand and a value of one to the first operand. The use of the one’s complement and the value of one instead of the two’s complement of the second operand results in a carry when the second operand is zero.

2. SUBTRACT LOGICAL differs from SUBTRACT only in the meaning of the condition code and in the absence of the interruption for overflow.

3. A zero difference is always accompanied by a carry out of bit position 0 for SLGR, SLGFR, SLG, and SLGF or bit position 32 for SLR, SL, and SLY, and, therefore, no borrow.

4. The condition-code setting for SUBTRACT LOGICAL can also be interpreted as indicating the presence or absence of a carry, as follows:
   1. Result not zero; no carry
   2. Result zero; carry
   3. Result not zero; carry

**SUBTRACT LOGICAL WITH BORROW**

SLBR  R1,R2  [RRE]

<table>
<thead>
<tr>
<th>'B999'</th>
<th>/</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

SLBGR  R1,R2  [RRE]

<table>
<thead>
<tr>
<th>'B989'</th>
<th>/</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

SLB  R1,Dz(Xz,Bz)  [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>R1</th>
<th>Xz</th>
<th>Bz</th>
<th>DLz</th>
<th>DHz</th>
<th>'99'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>
The second operand and the borrow are subtracted from the first operand, and the difference is placed at the first-operand location. For SUBTRACT LOGICAL WITH BORROW (SLBR, SLB), the operands, the borrow, and the difference are treated as 32-bit unsigned binary integers. For SUBTRACT LOGICAL WITH BORROW (SLBGR, SLBG), they are treated as 64-bit unsigned binary integers.

**Resulting Condition Code:**

0  Result zero; borrow  
1  Result not zero; borrow  
2  Result zero; no borrow  
3  Result not zero; no borrow  

**Program Exceptions:**

- Access (fetch, operand 2 of SLB and SLBG only)

**Programming Notes:**

1. A borrow is represented by a zero value of bit 18 of the current PSW. Bit 18 is the leftmost bit of the two-bit condition code in the PSW. Bit 18 is set to zero by an execution of a SUBTRACT LOGICAL or SUBTRACT LOGICAL WITH BORROW instruction that produces a borrow into the leftmost bit position of the 32-bit or 64-bit result.

2. Logical subtraction with borrow is performed by adding the one's complement of the second operand and bit 18 of the current PSW to the first operand. Therefore, when bit 18 is one, indicating no borrow, the addition is the same as for SUBTRACT LOGICAL.

3. Condition code zero is set for SUBTRACT LOGICAL WITH BORROW (SLBR, SLB), when the maximum 32-bit unsigned binary integer, $2^{32} - 1$, is subtracted from zero when PSW bit 18 indicates a borrow. For SUBTRACT LOGICAL WITH BORROW (SLBGR, SLBG) condition code zero is set when the maximum 64-bit unsigned binary integer, $2^{64} - 1$, is subtracted from zero when PSW bit 18 indicates a borrow.

4. SUBTRACT and SUBTRACT LOGICAL may provide better performance than SUBTRACT LOGICAL WITH BORROW, depending on the model.

**SUPERVISOR CALL**

<table>
<thead>
<tr>
<th>SVC I</th>
</tr>
</thead>
<tbody>
<tr>
<td>'OA' I</td>
</tr>
</tbody>
</table>

The instruction causes a supervisor-call interruption, with the I field of the instruction providing the rightmost byte of the interruption code.

Bits 8-15 of the instruction, with eight zeros appended on the left, are placed in the supervisor-call interruption code that is stored in the course of the interruption. See "Supervisor-Call Interruption" on page 6-47.

A serialization and checkpoint-synchronization function is performed.

**Condition Code:** The code remains unchanged and is saved as part of the old PSW. A new condition code is loaded as part of the supervisor-call interruption.

**Program Exceptions:** None.

**TEST ADDRESSING MODE**

<table>
<thead>
<tr>
<th>TAM [E]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'010B'</td>
</tr>
</tbody>
</table>

The extended-addressing-mode bit and basic-addressing-mode bit, bits 31 and 32 of the current PSW, respectively, are tested, and the result is indicated in the condition code.
**Resulting Condition Code:**

0 PSW bits 31 and 32 zeros (indicating 24-bit addressing mode)
1 PSW bit 31 zero and bit 32 one (indicating 31-bit addressing mode)
2 --
3 PSW bits 31 and 32 ones (indicating 64-bit addressing mode)

**Program Exceptions:** None.

**Programming Note:** The case when PSW bit 31 is one and bit 32 is zero causes an early PSW specification exception to be recognized.

---

**TEST AND SET**

TS D₂(B₂) [S]

<table>
<thead>
<tr>
<th>'93'</th>
<th>111111</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
</table>

The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the byte is set to all ones.

Bits 8-15 of the instruction are ignored.

The byte in storage is set to all ones as it is fetched for the testing of bit 0. This update appears to be an interlocked-update reference as observed by other CPUs.

A serialization function is performed before the byte is fetched and again after the storing of all ones.

**Resulting Condition Code:**

0 Leftmost bit zero
1 Leftmost bit one
2 --
3 --

**Program Exceptions:**

- Access (fetch and store, operand 2)
A mask is used to select bits of the first operand, and the result is indicated in the condition code.

TEST UNDER MASK is a new name of, and TMLH and TMLL are new mnemonics for, the ESA/390 instructions TEST UNDER MASK HIGH (TMH) and TEST UNDER MASK LOW (TML), respectively.

In TEST UNDER MASK (TM, TMY), the byte of immediate data, I₂, is used as an eight-bit mask. The bits of the mask are made to correspond one for one with the bits of the byte in storage designated by the first-operand address.

A mask bit of one indicates that the storage bit is to be tested. When the mask bit is zero, the storage bit is ignored. When all storage bits thus selected are zero, condition code 0 is set. Condition code 0 is also set when the mask is all zeros. When the selected bits are mixed zeros and ones, condition code 1 is set if the leftmost selected bit is zero, or condition code 2 is set if the leftmost selected bit is one. When the selected bits are all ones, condition code 3 is set.

Access exceptions associated with the storage operand are recognized for one byte even when the mask is all zeros.

In TEST UNDER MASK (TMHH, TMHL, TMLH, TMLL), the contents of the I₂ field are used as a 16-bit mask. For each instruction, the bits of the mask are made to correspond one for one with 16 bits of the first operand as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bits Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMHH</td>
<td>0-15</td>
</tr>
<tr>
<td>TMHL</td>
<td>16-31</td>
</tr>
<tr>
<td>TMLH (or TMH)</td>
<td>32-47</td>
</tr>
<tr>
<td>TMLL (or TML)</td>
<td>48-63</td>
</tr>
</tbody>
</table>

The displacement for TM is treated as a 12-bit unsigned binary integer. The displacement for TMY is treated as a 20-bit signed binary integer.

**Resulting Condition Code:**

0  Selected bits all zeros; or mask bits all zeros
1  Selected bits mixed zeros and ones (TM and TMY only)
2  Selected bits mixed zeros and ones, and leftmost is zero (TMHH, TMHL, TMLH, TMLL) -- (TM and TMY only)
2  Selected bits mixed zeros and ones, and leftmost is one (TMHH, TMHL, TMLH, TMLL)
3  Selected bits all ones

**Program Exceptions:**

- Access (fetch, operand 1, TM and TMY only)
- Operation (TMY, if the long-displacement facility is not installed)

**Programming Notes:**

1. An example of the use of the TEST UNDER MASK (TM) instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. When the mask for TMHH, TMHL, TMLH, or TMLL selects exactly two bits, the two selected bits effectively are loaded into the condition code.

**TRANSLATE**

<table>
<thead>
<tr>
<th>TR</th>
<th>D₁(L,B₁), D₂(B₂)</th>
<th>[SS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'DC'</td>
<td>L</td>
<td>B₁</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

The bytes of the first operand are used as eight-bit arguments to reference a list designated by the second-operand address. Each function
byte selected from the list replaces the corresponding argument in the first operand.

The L field specifies the length of only the first operand.

The bytes of the first operand are selected one by one for translation, proceeding left to right. Each argument byte is added to the initial second-operand address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an eight-bit unsigned binary integer and extended with zeros on the left. The sum is used as the address of the function byte, which then replaces the original argument byte.

The operation proceeds until the first-operand field is exhausted. The list is not altered unless an overlap occurs.

When the operands overlap, the result is obtained as if each result byte were stored immediately after fetching the corresponding function byte.

Access exceptions are recognized only for those bytes in the second operand which are actually required.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2; fetch and store, operand 1)

**Programming Notes:**
1. An example of the use of the TRANSLATE instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
2. TRANSLATE may be used to convert data from one code to another code.
3. The instruction may also be used to rearrange data. This may be accomplished by placing a pattern in the destination area, by designating the pattern as the first operand of TRANSLATE, and by designating the data that is to be rearranged as the second operand. Each byte of the pattern contains an eight-bit number specifying the byte destined for this position. Thus, when the instruction is executed, the pattern selects the bytes of the second operand in the desired order.
4. Because each eight-bit argument byte is added to the initial second-operand address to obtain the address of a function byte, the list may contain 256 bytes. In cases where it is known that not all eight-bit argument values will occur, it is possible to reduce the size of the list.
5. Significant performance degradation is possible when, with DAT on, the second-operand address of TRANSLATE designates a location that is less than 256 bytes to the left of a 4K-byte boundary. This is because the machine may perform a trial execution of the instruction to determine if the second operand actually crosses the boundary.
6. The fetch and subsequent store accesses to a particular byte in the first-operand field do not necessarily occur one immediately after the other. Thus, this instruction cannot be safely used to update a location in storage if the possibility exists that another CPU or a channel program may also be updating the location. An example of this effect is shown for OR (OI) in "Multiprogramming and Multiprocessing Examples" in Appendix A, "Number Representation and Instruction-Use Examples" on page A-1.
7. The storage-operand references of TRANSLATE may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

**TRANSLATE AND TEST**

```
<table>
<thead>
<tr>
<th>DD</th>
<th>L</th>
<th>B1</th>
<th>D2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>
```

The bytes of the first operand are used as eight-bit arguments to select function bytes from a list designated by the second-operand address. The first nonzero function byte is inserted in general register 2, and the related argument address in general register 1.

The L field specifies the length of only the first operand.

The bytes of the first operand are selected one by one for translation, proceeding left to right. The
first operand remains unchanged in storage. Calculation of the address of the function byte is performed as in the TRANSLATE instruction. The function byte retrieved from the list is inspected for a value of zero.

When the function byte is zero, the operation proceeds with the next byte of the first operand. When the first-operand field is exhausted before a nonzero function byte is encountered, the operation is completed by setting condition code 0. The contents of general registers 1 and 2 remain unchanged.

When the function byte is nonzero, the operation is completed by inserting the function byte in general register 2 and the related argument address in general register 1. This address points to the argument byte last translated. The function byte replaces bits 56-63 of general register 2, and bits 0-55 of this register remain unchanged. In the 24-bit addressing mode, the address replaces bits 40-63 of general register 1, and bits 0-39 of this register remain unchanged. In the 31-bit addressing mode, the address replaces bits 33-63 of general register 1, bit 32 of this register is set to zero, and bits 0-31 of the register remain unchanged. In the 64-bit addressing mode, the address replaces bits 0-63 of general register 1.

When the function byte is nonzero, either condition code 1 or 2 is set, depending on whether the argument byte is the rightmost byte of the first operand. Condition code 1 is set if one or more argument bytes remain to be translated. Condition code 2 is set if no more argument bytes remain.

The contents of access register 1 always remain unchanged.

Access exceptions are recognized only for those bytes in the second operand which are actually required. Access exceptions are not recognized for those bytes in the first operand which are to the right of the first byte for which a nonzero function byte is obtained.

**Resulting Condition Code:**

- 0: All function bytes zero
- 1: Nonzero function byte; first-operand field not exhausted
- 2: Nonzero function byte; first-operand field exhausted

**Program Exceptions:**

- Access (fetch, operands 1 and 2)

**Programming Notes:**

1. An example of the use of the TRANSLATE AND TEST instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. TRANSLATE AND TEST may be used to scan the first operand for characters with special meaning. The second operand, or list, is set up with all-zero function bytes for those characters to be skipped over and with nonzero function bytes for the characters to be detected.

**TRANSLATE AND TEST REVERSE**

```
TRTR D1(L,B1),D2(B2) [SS]
```

```
<table>
<thead>
<tr>
<th>'D0'</th>
<th>L</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>
```

The bytes of the first operand are used as eight-bit arguments to select function bytes from a list designated by the second-operand address. The first-operand address designates the rightmost byte of the first operand. The first nonzero function byte is inserted in general register 2, and the related argument address in general register 1.

The L field specifies the length of only the first operand.

The bytes of the first operand are selected one by one for translation, proceeding from right to left. The first operand remains unchanged in storage. Calculation of the address of the function byte is performed as in the TRANSLATE instruction. The function byte retrieved from the list is inspected for a value of zero.

When the function byte is zero, the operation proceeds with the preceding byte of the first operand. When the first-operand field is exhausted before a nonzero function byte is encountered, the operation is completed by setting condition code 0.
contents of general registers 1 and 2 remain unchanged.

When the function byte is nonzero, the operation is completed by inserting the function byte in general register 2 and the related argument address in general register 1. This address points to the argument byte last translated. The function byte replaces bits 56-63 of general register 2, and bits 0-55 of this register remain unchanged. In the 24-bit addressing mode, the address replaces bits 40-63 of general register 1, and bits 0-39 of this register remain unchanged. In the 31-bit addressing mode, the address replaces bits 33-63 of general register 1, and bits 0-32 of this register remain unchanged. In the 64-bit addressing mode, the address replaces bits 0-63 of general register 1.

When the function byte is nonzero, either condition code 1 or 2 is set, depending on whether the argument byte is the leftmost byte of the first operand. Condition code 1 is set if one or more argument bytes remain to be translated. Condition code 2 is set if no more argument bytes remain.

The contents of access register 1 always remain unchanged.

Access exceptions can be recognized for any byte in either the first or in the second operand.

**Resulting Condition Code:**

| 0 | All function bytes zero |
| 1 | Nonzero function byte; first-operand field not exhausted |
| 2 | Nonzero function byte; first-operand field exhausted |
| 3 | -- |

**Program Exceptions:**

- Access (fetch, operands 1 and 2)
- Operation (if the extended-translation facility 3 is not installed)

**Programming Note:**

1. TRANSLATE AND TEST REVERSE may be used to scan the first operand for characters with special meaning. The second operand, or list, is set up with all-zero function bytes for those characters to be skipped over and with nonzero function bytes for the characters to be detected.

**TRANSLATE EXTENDED**

TRE \text{R_1,R_2} \text{[RRE]}

<table>
<thead>
<tr>
<th>\text{'B2A5'}</th>
<th>\text{////////}</th>
<th>R_1</th>
<th>R_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The bytes of the first operand are compared to a test byte in general register 0 and, unless an equal comparison occurs, are used as eight-bit arguments to reference a 256-byte translation table designated by the second-operand address. Each function byte selected from the second operand replaces the corresponding argument in the first operand. The operation proceeds until a first-operand byte equal to the test byte is encountered, the end of the first operand is reached, or a CPU-determined number of bytes have been processed, whichever occurs first. The result is indicated in the condition code.

The \text{R_1} field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers \text{R_1} and \text{R_2}, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the first-operand location is specified by the contents of bit positions 32-63 of general register \text{R_1+1}, and those contents are treated as a 32-bit unsigned binary integer. In the 64-bit addressing mode, the number of bytes in the first-operand location is specified by the entire contents of general register \text{R_1+1}, and those contents are treated as a 64-bit unsigned binary integer.

The handling of the addresses in general registers \text{R_1} and \text{R_2} is dependent on the addressing mode.
In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers R₁ and R₂ constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of the registers constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 constitute the address.

The test byte is in bit positions 56-63 of general register 0, and the contents of bit positions 0-55 of this register are ignored.

The contents of the registers just described are shown in Figure 7-79.

The bytes of the first operand are selected one by one for translation, proceeding left to right. Each argument byte is first compared to the test byte in general register 0. If the result is an equal comparison, the operation is completed. If the argument byte is not equal to the test byte, the argument byte is added to the initial second-operand address. The addition is performed following the rules for address arithmetic, with the argument byte treated as an eight-bit unsigned binary integer and extended with zeros on the left. The
sum is used as the address of the function byte, which then replaces the original argument byte. The second operand is not altered unless an overlap occurs.

The operation proceeds until a first-operand byte equal to the test byte is encountered, the first-operand location is exhausted, or a CPU-determined number of first-operand bytes have been processed.

When the first-operand location is exhausted without finding a byte equal to the test byte, condition code 0 is set. When a first-operand byte equal to the test byte is encountered, condition code 1 is set. When a CPU-determined number of bytes have been processed, condition code 3 is set. Condition code 3 may be set even when the first-operand location is exhausted or when the next byte to be processed is equal to the test byte. In these cases, condition code 0 or 1, respectively, will be set when the instruction is executed again.

If the operation is completed with condition code 0, the contents of general register \( R_1 \) are incremented by the contents of general register \( R_1 + 1 \), and then the contents of general register \( R_1 + 1 \) are set to zero. If the operation is completed with condition code 1, the contents of general register \( R_1 + 1 \) are decremented by the number of bytes processed before the first-operand byte equal to the test byte was encountered, and the contents of general register \( R_1 \) are incremented by the same number, so that general register \( R_2 \) contains the address of the equal byte. If the operation is completed with condition code 3, the contents of general register \( R_1 + 1 \) are decremented by the number of bytes processed, and the contents of general register \( R_1 \) are incremented by the same number, so that the instruction, when reexecuted, resumes at the next byte to be processed. When general register \( R_1 \) is updated in the 24-bit or 31-bit addressing mode, bits 32-39 of it, in the 24-bit mode, or bit 32, in the 31-bit mode, may be set to zeros or may remain unchanged from their original values.

In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers \( R_1 \) and \( R_1 + 1 \) always remain unchanged.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.

When the \( R_2 \) register is the same register as the \( R_1 \) or \( R_1 + 1 \) register, the results are unpredictable.

When \( R_1 \) or \( R_2 \) is zero, the results are unpredictable.

When the second operand overlaps the first operand, the results are unpredictable.

Access exceptions for the portion of the first operand to the right of the last byte processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last byte processed.

Access exceptions for all 256 bytes of the second operand may be recognized, even if not all bytes are used.

Access exceptions are not recognized if the \( R_1 \) field is odd. When the length of the first operand is zero, no access exceptions for the first operand are recognized.

Resulting Condition Code:

0 Entire first operand processed without finding a byte equal to the test byte
1 First-operand byte is equal to the test byte
2 --
3 CPU-determined number of bytes processed

Program Exceptions:
- Access (fetch, operand 2; store, operand 1)
- Specification

Programming Notes:
1. When condition code 3 is set, the program can simply branch back to the instruction to continue the translation. The program need not determine the number of bytes that were translated.

2. The instruction can improve performance by being used in place of a TRANSLATE AND TEST instruction that locates an escape character, followed by a TRANSLATE instruction that translates the bytes preceding the escape character.
3. The storage-operand references of TRANSLATE EXTENDED may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98.)

TRANSLATE ONE TO ONE

TR00 R1,R2 [RRE]

'0993' // // // // R1 R2
0 16 24 28 31

TRANSLATE ONE TO TWO

TR0T R1,R2 [RRE]

'0992' // // // // R1 R2
0 16 24 28 31

TRANSLATE TWO TO ONE

TRTO R1,R2 [RRE]

'0991' // // // // R1 R2
0 16 24 28 31

TRANSLATE TWO TO TWO

TRTT R1,R2 [RRE]

'0990' // // // // R1 R2
0 16 24 28 31

The characters of the second operand are used as arguments to select function characters from a translation table designated by the address in general register 1. Each function character selected from the translation table is compared to a test character in general register 0, and, unless an equal comparison occurs, is placed at the first-operand location. The operation proceeds until a selected function character equal to the test character is encountered, the end of the second operand is reached, or a CPU-determined number of characters have been processed, whichever occurs first. The result is indicated in the condition code.

The lengths of the operand and test characters are as follows:

- For TRANSLATE ONE TO ONE, the second-operand, first-operand, and test characters are single bytes.
- For TRANSLATE ONE TO TWO, the second-operand characters are single bytes, and the first-operand and test characters are double bytes.
- For TRANSLATE TWO TO ONE, the second-operand characters are double bytes, and the first-operand and test characters are single bytes.
- For TRANSLATE TWO TO TWO, the second-operand, first-operand, and test characters are double bytes.

For TRANSLATE ONE TO ONE and TRANSLATE TWO TO ONE, the test character is in bit positions 56-63 of general register 0. For TRANSLATE ONE TO TWO and TRANSLATE TWO TO TWO, the test character is in bit positions 48-63 of general register 0.

The R1 field designates an even-odd pair of general registers and must designate an even-numbered register; otherwise, a specification exception is recognized.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R1 and R2, respectively. In the 24-bit or 31-bit addressing mode, the number of bytes in the second-operand location is specified by the contents of bit positions 32-63 of general register R1 + 1, and those contents are treated as a 32-bit unsigned binary integer. In the 64-bit addressing mode, the number of bytes in the second-operand location is specified by the contents of bit positions 0-63 of general register R1 + 1, and those contents are treated as a 64-bit unsigned binary integer. The length of the first-operand location is considered to be the same as that of the second operand for TRANSLATE ONE TO ONE and TRANSLATE TWO TO TWO, twice that for TRANSLATE ONE TO TWO, and one half that for TRANSLATE TWO TO ONE.

For TRANSLATE TWO TO ONE and TRANSLATE TWO TO TWO, the length in general register R1 + 1 must be an even number of bytes; otherwise, a specification exception is recognized.
The translation table is treated as being on a doubleword boundary for TRANSLATE ONE TO ONE and TRANSLATE ONE TO TWO and on a 4K-byte boundary for TRANSLATE TWO TO ONE and TRANSLATE TWO TO TWO. The rightmost bits of the register that are not used to form the address, which are bits 61-63 in the doubleword case and bits 52-63 in the 4K-byte case, are ignored.

The handling of the addresses in general registers $R_1$, $R_2$, and 1 is dependent on the addressing mode.

In the 24-bit addressing mode, the contents of bit positions 40-63 of general registers $R_1$ and $R_2$ and 40-60 or 40-51 of 1 constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-63 of registers $R_1$ and $R_2$ and 33-60 or 33-51 of 1 constitute the address, and the contents of bit positions 0-32 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-63 of registers $R_1$ and $R_2$ and 0-60 or 0-51 of 1 constitute the address.

The contents of the registers just described are shown in Figure 7-80.

Figure 7-80 (Part 1 of 2). Register Contents for TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO
In the access-register mode, the contents of access registers \( R_1, R_2, \) and 1 are used for accessing the first operand, second operand, and translation table, respectively.

The length of the translation table designated by the address contained in general register 1 is as follows:

- For TRANSLATE ONE TO ONE, the translation-table length is 256 bytes; each of the 256 function characters is a single byte.
- For TRANSLATE ONE TO TWO, the translation-table length is 512 bytes; each of the 256 function characters is a double byte.
- For TRANSLATE TWO TO ONE, the translation-table length is 65,536 (64K) bytes; each of the 64K function characters is a single byte.
- For TRANSLATE TWO TO TWO, the translation-table length is 131,072 (128K) bytes; each of the 64K function characters is a double byte.

The characters of the second operand are selected one by one for translation, proceeding left to right. Each argument character is added to the initial translation-table address. The addition is performed following the rules for address arithmetic, with the argument character treated as follows:

- For TRANSLATE ONE TO ONE, the argument character is treated as an eight-bit unsigned binary integer extended on the left with 56 zeros.
- For TRANSLATE ONE TO TWO, the argument character is treated as an eight-bit unsigned binary integer extended on the right with a zero and on the left with 55 zeros.
- For TRANSLATE TWO TO ONE, the argument character is treated as a 16-bit unsigned

![64-Bit Addressing Mode Diagram](image)

Figure 7-80 (Part 2 of 2). Register Contents for TRANSLATE ONE TO ONE, TRANSLATE ONE TO TWO, TRANSLATE TWO TO ONE, and TRANSLATE TWO TO TWO
binary integer extended on the left with 48 zeros.

- For TRANSLATE TWO TO TWO, the argument character is treated as a 16-bit unsigned binary integer extended on the right with a zero and on the left with 47 zeros.

The rightmost bits of the translation-table address that are ignored (61-63 or 52-63) are treated as zeros during this addition.

The sum is used as the address of the function character.

Each function character selected as described above is first compared to the test character in general register 0. If the result is an equal comparison, the operation is completed. If the function character is not equal to the test character, the function character is placed in the next available character position in the first operand, that is, the first function character is placed at the beginning of the first-operand location, and each successive function character is placed immediately to the right of the preceding character. The second operand and the translation table are not altered unless an overlap occurs.

The operation proceeds until a selected function character equal to the test character is encountered, the second-operand location is exhausted, or a CPU-determined number of second-operand characters have been processed.

When a selected function character equal to the test character is encountered, condition code 1 is set. When the second-operand location is exhausted without finding a selected function character equal to the test character, condition code 0 is set. When a CPU-determined number of characters have been processed, condition code 3 is set. Condition code 3 may be set even when the next character to be processed results in a function character equal to the test character or when the second-operand location is exhausted. In these cases, condition code 1 or 0, respectively, will be set when the instruction is executed again.

If the operation is completed with condition code 0, the contents of general register R₂ are incremented by the contents of general register R₁ + 1, and the contents of general register R₁ are incremented as follows:

- For TRANSLATE ONE TO ONE and TRANSLATE TWO TO TWO, the same as for general register R₂.
- For TRANSLATE ONE TO TWO, by twice the amount for general register R₂.
- For TRANSLATE TWO TO ONE, by one half the amount for general register R₂.

The contents of general register R₁ + 1 are then set to zero.

If the operation is completed with condition code 1, the contents of general register R₁ + 1 are decremented by the number of second-operand bytes processed before the character that selected a function character equal to the test character was encountered, and the contents of general register R₂ are incremented by the same number, so that general register R₂ contains the address of the character that selected a function character equal to the test character. The contents of general register R₁ are incremented by the same, twice, or one half the number, as described above for condition code 0.

If the operation is completed with condition code 3, the contents of general register R₁ + 1 are decremented by the number of second-operand bytes processed, and the contents of general register R₂ are incremented by the same number, so that the instruction, when reexecuted, contains the address of the next character to be processed. The contents of general register R₁ are incremented by the same, twice, or one half the number, as described above for condition code 0.

When general registers R₁ and R₂ are updated in the 24-bit or 31-bit addressing mode, the bits in bit positions 32-39 of them that are not part of the address may be set to zeros or may remain unchanged from their original values. In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general registers R₁, R₁ + 1, and R₂ always remain unchanged.

The contents of general registers 0 and 1 remain unchanged.

The amount of processing that results in the setting of condition code 3 is determined by the CPU on the basis of improving system performance, and it may be a different amount each time the instruction is executed.
During instruction execution, CPU retry may result in condition code 3 being set with possibly incorrect data having been stored in the first operand location at or to the right of the location designated by the final address in general register R1. The amount of data stored depends on the operation and the point in time at which CPU retry occurred. In all cases, the storing will occur again, with correct data stored, when the instruction is executed again to continue processing the same operands.

When the R1 register is the same register as the R2 register, the R1 or R2 register is register 0, or the R2 register is register 1, the results are unpredictable.

When any of the first and second operands and the translation table overlaps another of them, the results are unpredictable.

Access exceptions for the portion of the first or second operand to the right of the last character processed may or may not be recognized. For an operand longer than 4K bytes, access exceptions are not recognized for locations more than 4K bytes beyond the last character processed.

Access exceptions for all characters of the translation table may be recognized even if not all characters are used.

Access exceptions are not recognized if the R1 field is odd. When the length of the second operand is zero, no access exceptions for the first or second operand are recognized, and access exceptions for the translation table may or may not be recognized.

Resulting Condition Code:

0  Entire second operand processed without finding a resulting function character equal to the test character
1  Second-operand character found resulting in a function character equal to the test character
2  --
3  CPU-determined number of characters processed

Program Exceptions:

- Operation (if the extended-translation facility 2 is not installed)
- Specification

Programming Notes:

1. These instructions differ from the TRANSLATE EXTENDED instruction by having the following attributes:
   - Depending on the instruction used, the sets of argument characters and function characters each can contain single-byte or double-byte characters.
   - The test character is compared to a resulting function character instead of to an argument character.
   - The argument (source) and function (destination) operands are different operands.

2. When condition code 3 is set, the program can simply branch back to the instruction to continue the translation. The program need not determine the number of characters that were translated.

3. The storage operand references of these instructions may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

UNPACK

UNPK  D1(L1,B1),D2(L2,B2)  [SS]

<table>
<thead>
<tr>
<th>'F3'</th>
<th>L1</th>
<th>L2</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
<td>47</td>
</tr>
</tbody>
</table>

The format of the second operand is changed from packed to zoned, and the result is placed at the first-operand location. The packed and zoned formats are described in Chapter 8, “Decimal Instructions.”

The second operand is treated as having the packed format. Its digits and sign are placed unchanged in the first-operand location, using the zoned format. Zone bits with coding of 1111 are supplied for all bytes except the rightmost byte, the zone of which receives the sign of the second operand. The sign and digits are not checked for valid codes.
The result is obtained as if the operands were processed right to left. When necessary, the second operand is considered to be extended on the left with zeros. If the first-operand field is too short to contain all digits of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the operands overlap, the result is obtained as if the operands were processed one byte at a time and as if the first result byte were stored immediately after fetching the first operand byte. The entire rightmost second-operand byte is used in forming the first result byte. For the remainder of the field, information for two result bytes is obtained from a single second-operand byte, and execution proceeds as if the leftmost four bits of the byte were to remain available for the next result byte and need not be refetched. Thus, the result is as if two result bytes were to be stored immediately after fetching a single operand byte.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2; store, operand 1)

**Programming Notes:**
1. An example of the use of the UNPACK instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. A field that is to be unpacked can be destroyed by improper overlapping. To save storage space for unpacking by overlapping the operands, the rightmost byte of the first operand must be to the right of the rightmost byte of the second operand by the number of bytes in the second operand minus 2. If only one or two bytes are to be unpacked, the rightmost bytes of the two operands may coincide.
3. The storage-operand references of UNPACK may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

### UNPACK ASCII

UNPACK ASCII

```
UNPKA D1(L1,B1),D2(B2) [SS]
```

<table>
<thead>
<tr>
<th>'EA'</th>
<th>L1</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>
```

The format of the second operand is changed from packed to ASCII, and the result is placed at the first-operand location. The packed format is described in Chapter 8, “Decimal Instructions.”

The second operand is treated as having the packed format. Its digits are converted to ASCII characters by extending them on the left with 0011 binary, and the ASCII characters are then placed at the first operand location. The digits are not checked for valid codes.

The sign of the second operand is not transferred to the first operand but is checked for validity and determines the condition code. If the sign is 1010, 1100, 1110 or 1111 binary (plus), condition code 0 is set. If the sign is 1011 or 1101 binary (minus), condition code 1 is set. If the sign is not one of the codes for plus or minus, condition code 3 is set.

The converted last digit is placed in the rightmost byte position of the result field, and the other converted digits are placed adjacent to the last and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left.

The length of the second operand is 16 bytes. The second operand consists of 31 digits and a sign.

The length of the first operand is designated by the contents of the L1 field. The first-operand length must not exceed 32 bytes (L1 must be less than or equal to 31); otherwise, a specification exception is recognized.

If the first operand is too short to contain all digits of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.
When the length of the first operand is 32 bytes, the leftmost byte is set to ASCII zero, 30 hex.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first operand is not necessarily stored into in any particular order.

**Resulting Condition Code:**

0  Sign is plus
1  Sign is minus
2  --
3  Sign is invalid

**Program Exceptions:**

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility is not installed)
- Specification

**Programming Note:**

1. The following example illustrates the use of the instruction to unpack to ASCII digits:

   ```
   ASDIGITS DS CL31
   PKDIGITS DS OPL16
   DC X'12345678901'
   DC X'12345678901'
   DC X'12345678901'
   DC X'1C'
   ...
   UNPKA ASDIGITS(31),PKDIGITS
   ```

2. The storage-operand references of UNPACK ASCII may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

**UNPACK UNICODE**

The second operand is treated as having the packed format. Its digits are converted to two-byte Unicode characters by extending them on the left with 0000000000011 binary (003 hex), and the Unicode characters are then placed at the first operand location. The digits are not checked for valid codes. The sign of the second operand is not transferred to the first operand but is checked for validity and determines the condition code. If the sign is 1010, 1100, 1110 or 1111 binary (plus), condition code 0 is set. If the sign is 1011 or 1101 binary (minus), condition code 1 is set. If the sign is not one of the codes for plus or minus, condition code 3 is set.

The converted last digit is placed in the rightmost character position of the result field, and the other converted digits are placed adjacent to the last and to each other in the remainder of the result field.

The result is obtained as if the operands were processed right to left.

The length of the second operand is 16 bytes; the second operand consists of 31 digits and a sign.

If the first operand is too short to contain all digits of the second operand, the remaining leftmost portion of the second operand is ignored. Access exceptions for the unused portion of the second operand may or may not be indicated.

When the length of the first operand is 32 characters, the leftmost character is set to Unicode Basic Latin zero, 0030 hex.

The results are unpredictable if the first and second operands overlap in any way.

As observed by other CPUs and by channel programs, the first operand is not necessarily stored into in any particular order.

**Resulting Condition Code:**

0  Sign is plus
1  Sign is minus
Sign is invalid

Program Exceptions:

- Access (fetch, operand 2; store, operand 1)
- Operation (if the extended-translation facility 2 is not installed)
- Specification

Programming Notes:

1. The following example illustrates the use of the instruction to unpack to European numbers:

   UNDIGITS DS CL62
   PKDIGITS DS 0PL16
   DC X'1234567890'
   DC X'1234567890 '
   DC X'1234567890 '
   DC X'1C'
   ...
   UNPKU UNDIGITS(62),PKDIGITS

2. The storage-operand references of UNPACK UNICODE may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98.)

UPDATE TREE

The nodes of a tree in storage are examined successively on a path toward the base of the tree, and contents of general register 0, conceptually followed on the right by contents of general register 1, are conditionally interchanged with the contents of the nodes so as to give a unique maximum logical value in general register 0. The first half of a node and general register 0 contain a codeword, which is for use in sort/merge algorithms.

If the codeword in general register 0 equals the codeword in a node, the contents of the node are placed in general registers 2 and 3.

General register 4 contains the base address of the tree, and general register 5 contains the index of a node whose parent node will be examined first.

In the access-register mode, access register 4 specifies the address space containing the tree.

This instruction may be interrupted between units of operation. The condition code is unpredictable if the instruction is interrupted.

The size of a node, the size of a codeword, and the participation of bits 0-31 of general registers 1-5 in the operation depend on the addressing mode. In the 24-bit or 31-bit addressing mode, a node is eight bytes, a codeword is four bytes, and bits 0-31 are ignored and remain unchanged. In the 64-bit addressing mode, a node is 16 bytes, a codeword is eight bytes, and bits 0-31 are used in and may be changed by the operation.

Operation in the 24-Bit or 31-Bit Addressing Mode

In the 24-bit or 31-bit addressing mode, the doubleword nodes of a tree in storage are examined successively on a path toward the base of the tree, and the contents of bit positions 32-63 of general register 0, conceptually followed on the right by the contents of bit positions 32-63 of general register 1, are conditionally interchanged with the contents of the nodes so as to give a unique maximum logical value in bit positions 32-63 of general register 0.

Bit positions 32-63 of general register 4 contain the base address of the tree, and bit positions 32-63 of general register 5 contain the index of a node whose parent node will be examined first. The base address is eight less than the address of the root node of the tree. The initial contents of bit positions 32-63 of general registers 4 and 5 must be a multiple of 8; otherwise, a specification exception is recognized.

A unit of operation begins by shifting the contents of bit positions 32-63 of general register 5 right logically one position and then setting bit 61 to zero. However, bits 32-63 of general register 5 remain unchanged if the execution of a unit of operation is nullified or suppressed. If after shifting and setting bit 61 to zero, bits 32-63 of general register 5 are all zeros, the instruction is completed, and condition code 1 is set; otherwise, the unit of operation continues.
Bit 32 of general register 0 is tested. If bit 32 of general register 0 is one, the instruction is completed, and condition code 3 is set.

If bit 32 of general register 0 is zero, the sum of bits 32-63 of general registers 4 and 5 is used as the intermediate value for normal operand address generation. The generated address is the address of a node in storage.

Bits 32-63 of general register 0 are logically compared with the contents of the first word of the currently addressed node. If the register operand is low, the contents of bit positions 32-63 of general registers 0 and 1 are interchanged with those of the node, and a unit of operation is completed. If the register operand is high, no additional action is taken, and the unit of operation is completed. If the compare values are equal, bit positions 32-63 of general register 2, conceptually followed on the right by bit positions 32-63 of general register 3, are loaded from the currently addressed node, the instruction is completed, and condition code 0 is set.

In those cases when the value in the first word of the node is less than or equal to the value in bit positions 32-63 of the register, the contents of the node remain unchanged. However, in some models, these contents may be fetched and subsequently stored back.

**Operation in the 64-Bit Addressing Mode**

In the 64-bit addressing mode, the quadword nodes of a tree in storage are examined successively on a path toward the base of the tree, and the contents of general register 0, conceptually followed on the right by the contents of general register 1, are conditionally interchanged with the contents of the nodes so as to give a unique maximum logical value in general register 0.

General register 4 contains the base address of the tree, and general register 5 contain the index of a node whose parent node will be examined first. The base address is 16 less than the address of the root node of the tree. The initial contents of general registers 4 and 5 must be a multiple of 16; otherwise, a specification exception is recognized.

A unit of operation begins by shifting the contents of general register 5 right logically one position and then setting bit 60 to zero. However, general register 5 remains unchanged if the execution of a unit of operation is nullified or suppressed. If after shifting and setting bit 60 to zero, the contents of general register 5 are zero, the instruction is completed, and condition code 1 is set; otherwise, the unit of operation continues.

Bit 0 of general register 0 is tested. If bit 0 of general register 0 is one, the instruction is completed, and condition code 3 is set.

If bit 0 of general register 0 is zero, the sum of the contents of general registers 4 and 5 is used as the intermediate value for normal operand address generation. The generated address is the address of a node in storage.

The contents of general register 0 are logically compared with the contents of the first doubleword of the currently addressed node. If the register operand is low, the contents of general registers 0 and 1 are interchanged with those of the node, and a unit of operation is completed. If the register operand is high, no additional action is taken, and the unit of operation is completed. If the compare values are equal, general registers 2 and 3 are loaded from the currently addressed node, the instruction is completed, and condition code 0 is set.

In those cases when the value in the first doubleword of the node is less than or equal to the value in the register, the contents of the node remain unchanged. However, in some models, these contents may be fetched and subsequently stored back.

**Specifications Independent of Addressing Mode**

Access exceptions are recognized only for one node at a time. Access exceptions, change-bit action, and PER storage alteration do not occur for subsequent nodes until the previous node has been successfully compared and updated, and they also do not occur if a specification-exception condition exists.

**Resulting Condition Code:**

0  Equal compare values at currently addressed node
1  No equal compare values found on path, or no comparison made
In 24-bit or 31-bit mode, bits 32-63 of general register 5 nonzero and bits 32-63 of general register 0 negative; in 64-bit mode, general register 5 nonzero and general register 0 negative.

Program Exceptions:

- Access (fetch and store, nodes of tree)
- Specification

Programming Notes:

1. An example of the use of UPDATE TREE is given in "Sorting Instructions" in Appendix A, "Number Representation and Instruction-Use Examples."

2. For use in sorting in the 24-bit or 31-bit addressing mode, when equal compare values have been found, the contents of bit positions 32-63 of general registers 1 and 3 can be appropriate (depending on the contents of the tree) for the subsequent execution of COMPARE AND FORM CODEWORD. The contents of bit positions 32-63 of general register 2, shifted right 16 bit positions, can be similarly appropriate, and they can provide for minimal recomparison of partially equal keys. The same applies in the 64-bit addressing mode except to the contents of bit positions 0-63 of the registers and with the contents of bit positions 0-63 of general register 2 shifted right 48 bit positions. Refer to "Sorting Instructions" on page A-51 for a discussion of trees and their use in sorting.

3. The program should avoid placing a nonzero value in bit positions 32-38 of general register 5 when in the 24-bit addressing mode. If any bit in bit positions 32-38 is a one, the nodes of the tree will not be examined successively.

4. When bits 32-63 of general register 0 are negative in the 24-bit or 31-bit addressing mode, or when bits 0-63 are negative in the 64-bit mode, and provided that the tree has been updated properly previously, the node represented by general registers 0 and 1 either is the node or is equal to the node (equal keys) that would be selected if the unit of operation continued. In this case, ending the unit of operation and setting condition code 3 is a faster means of selecting an appropriate node because it does not require further examination and updating of the tree.

5. Setting condition code 3 provides improved performance when the replacement record is equal to the old winner and, more importantly (since the first case can be detected by means of the condition code of CFC), when the update path contains a negative codeword, indicating equality with the old winner.

6. In those cases when the codeword in the node is less than or equal to the codeword in general register 0, depending on the model, the contents of the node may be fetched and subsequently stored back. As a result, any of the following may occur for the storage location containing the node: a PER storage-alteration event may be recognized; a protection exception for storing may be recognized; and, provided no access exceptions exist, the change bit may be set to one. Because the contents of storage remain unchanged, the change bit may or may not be one when a PER storage-alteration event is recognized.

7. Special precautions should be taken when UPDATE TREE is made the target of EXECUTE. See the programming note concerning interruptible instructions under EXECUTE.

8. Further programming notes concerning interruptible instructions are included in "Interruptible Instructions" on page 5-21.

9. The storage-operand references for UPDATE TREE may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

10. Figure 7-81 on page 7-208 is a summary of the operation of UPDATE TREE in the 24-bit or 31-bit addressing mode, and Figure 7-82 on page 7-209 is a summary of the operation in the 64-bit addressing mode.
Figure 7-81. Execution of UPDATE TREE in the 24-Bit or 31-Bit Addressing Mode
Figure 7-82. Execution of UPDATE TREE in the 64-Bit Addressing Mode
Chapter 8. Decimal Instructions

Decimal-Number Formats .................................. 8-1
Zoned Format .............................................. 8-1
Packed Format ............................................. 8-1
Decimal Codes .............................................. 8-2
Decimal Operations ....................................... 8-2
Decimal-Arithmetic Instructions ......................... 8-2
Editing Instructions ...................................... 8-3
Execution of Decimal Instructions ....................... 8-3
Other Instructions for Decimal Operands ............... 8-3
Decimal-Operand Data Exception ......................... 8-4
Instructions ................................................ 8-4
ADD DECIMAL .............................................. 8-5
COMPARE DECIMAL ........................................ 8-6
DIVIDE DECIMAL .......................................... 8-6
EDIT ......................................................... 8-7
EDIT AND MARK ........................................... 8-9
MULTIPLY DECIMAL ....................................... 8-11
SHIFT AND ROUND DECIMAL ............................. 8-11
SUBTRACT DECIMAL ....................................... 8-12
TEST DECIMAL .............................................. 8-13
ZERO AND ADD ............................................ 8-13

The decimal instructions of this chapter perform arithmetic and editing operations on decimal data. Additional operations on decimal data are provided by several of the instructions in Chapter 7, "General Instructions." Decimal operands always reside in storage, and most decimal instructions use the SS instruction format. Decimal operands occupy storage fields that can start on any byte boundary.

Decimal-Number Formats

Decimal numbers may be represented in either the zoned or packed format. Both decimal-number formats are of variable length; the instructions used to operate on decimal data each specify the length of their operands and results. Each byte of either format consists of a pair of four-bit codes; the four-bit codes include decimal-digit codes, sign codes, and a zone code.

Zoned Format

In the zoned format, the rightmost four bits of a byte are called the numeric bits (N) and normally consist of a code representing a decimal digit. The leftmost four bits of a byte are called the zone bits (Z), except for the rightmost byte of a decimal operand, where these bits may be treated either as a zone or as a sign (S).

Decimal digits in the zoned format may be part of a larger character set, which includes also alphabetic and special characters. The zoned format is, therefore, suitable for input, editing, and output of numeric data in human-readable form. There are no decimal-arithmetic instructions which operate directly on decimal numbers in the zoned format; such numbers must first be converted to the packed format.

The editing instructions produce a result of up to 256 bytes; each byte may be a decimal digit in the zoned format, a message byte, or a fill byte.

Packed Format

In the packed format, each byte contains two decimal digits (D), except for the rightmost byte, which contains a sign to the right of a decimal digit. Decimal arithmetic is performed with operands in the packed format and generates results in the packed format.

The packed-format operands and results of decimal-arithmetic instructions may be up to 16 bytes (31 digits and sign), except that the maximum length of a multiplier or divisor is eight bytes (15 digits and sign). In division, the sum of the lengths of the quotient and remainder may be from two to 16 bytes. The editing instructions can fetch as many as 256 decimal digits from one or more decimal numbers of variable length, each in the packed format.
Decimal Codes

The decimal digits 0-9 have the binary encoding 0000-1001.

The preferred sign codes are 1100 for plus and 1101 for minus. These are the sign codes generated for the results of the decimal-arithmetic instructions and the CONVERT TO DECIMAL instruction.

Alternate sign codes are also recognized as valid in the sign position: 1010, 1110, and 1111 are alternate codes for plus, and 1011 is an alternate code for minus. Alternate sign codes are accepted for any decimal source operand, but are not generated in the completed result of a decimal-arithmetic instruction or CONVERT TO DECIMAL. This is true even when an operand remains otherwise unchanged, such as when adding zero to a number. An alternate sign code is, however, left unchanged by MOVE NUMERICS, MOVE WITH OFFSET, MOVE ZONES, PACK, and UNPACK.

When an invalid sign or digit code is detected, a data exception is recognized. For the decimal-arithmetic instructions and CONVERT TO BINARY, the operation is suppressed.

For the editing instructions EDIT and EDIT AND MARK, an invalid sign code is not recognized. The operation is terminated for a data exception due to an invalid digit code. No validity checking is performed by MOVE NUMERICS, MOVE WITH OFFSET, MOVE ZONES, PACK, and UNPACK.

The zone code 1111 is generated in the left four bit positions of each byte representing a zone and a decimal digit in zoned-format results. Zoned-format results are produced by EDIT, EDIT AND MARK, and UNPACK. For EDIT and EDIT AND MARK, each result byte representing a zoned-format decimal digit contains the zone code 1111 in the left four bit positions and the decimal-digit code in the right four bit positions. For UNPACK, zone bits with a coding of 1111 are supplied for all bytes except the rightmost byte, the zone of which receives the sign.

The meaning of the decimal codes is summarized in Figure 8-1.

<table>
<thead>
<tr>
<th>Code (Binary)</th>
<th>Recognized As</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Digit</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>Invalid</td>
</tr>
<tr>
<td>1011</td>
<td>Invalid</td>
</tr>
<tr>
<td>1100</td>
<td>Invalid</td>
</tr>
<tr>
<td>1101</td>
<td>Invalid</td>
</tr>
<tr>
<td>1110</td>
<td>Invalid</td>
</tr>
<tr>
<td>1111</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Figure 8-1. Summary of Digit and Sign Codes

Programming Note: Since 1111 is both the zone code and an alternate code for plus, unsigned (positive) decimal numbers may be represented in the zoned format with 1111 zone codes in all byte positions. The result of the PACK instruction converting such a number to the packed format may be used directly as an operand for decimal instructions.

Decimal Operations

The decimal instructions in this chapter consist of two classes, the decimal-arithmetic instructions and the editing instructions.

Decimal-Arithmetic Instructions

The decimal-arithmetic instructions perform addition, subtraction, multiplication, division, comparison, and shifting.

Operands of the decimal-arithmetic instructions are in the packed format and are treated as signed decimal integers. A decimal integer is represented in true form as an absolute value with a separate plus or minus sign. It contains an odd number of decimal digits, from one to 31, and the sign; this corresponds to an operand length of one to 16 bytes.

A decimal zero normally has a plus sign, but multiplication, division, and overflow may produce a
zero value with a minus sign. Such a negative zero is a valid operand and is treated as equal to a positive zero by COMPARE DECIMAL.

The lengths of the two operands specified in the instruction need not be the same. If necessary, the shorter operand is considered to be extended with zeros on the left. Results, however, cannot exceed the first-operand length as specified in the instruction.

When a carry or leftmost nonzero digits of the result are lost because the first-operand field is too short, the result is obtained by ignoring the overflow digits, condition code 3 is set, and, if the decimal-overflow mask bit is one, a program interruption for decimal overflow occurs. The operand lengths alone are not an indication of overflow; nonzero digits must have been lost during the operation.

The operands of decimal-arithmetic instructions should not overlap at all or should have coincident rightmost bytes. In ZERO AND ADD, the operands may also overlap in such a manner that the rightmost byte of the first operand (which becomes the result) is to the right of the rightmost byte of the second operand. For these cases of proper overlap, the result is obtained as if operands were processed right to left. Because the codes for digits and signs are verified during the performance of the arithmetic, improperly overlapping operands are recognized as data exceptions. However, in ZERO AND ADD when the rightmost byte of the first operand is to the left of the rightmost byte of the second operand, the entire second operand may be fetched, depending on the model, before any storing occurs, which will cause a data exception not to be recognized. See "Interlocks within a Single Instruction" on page 5-91 for how overlap is detected in the access-register mode.

Programming Note: A packed decimal number in storage may be designated as both the first and second operand of ADD DECIMAL, COMPARE DECIMAL, DIVIDE DECIMAL, MULTIPLY DECIMAL, SUBTRACT DECIMAL, or ZERO AND ADD. Thus, a decimal number may be added to itself, compared with itself, and so forth; SUBTRACT DECIMAL may be used to set a decimal field in storage to zero; and, for MULTIPLY DECIMAL, a decimal number may be squared in place. In these cases, the lengths of the two operands are not necessarily equal and may, depending on the instruction, be prohibited from being equal.

Editing Instructions

The editing instructions are EDIT and EDIT AND MARK. For these instructions, only the first operand (the pattern) has an explicitly specified length. The second operand (the source) is considered to have as many digits as necessary for the completion of the operation.

Overlapping operands for the editing instructions yield unpredictable results.

Execution of Decimal Instructions

During the execution of a decimal instruction, all bytes of the operands are not necessarily accessed concurrently, and the fetch and store accesses to a single location do not necessarily occur one immediately after the other. Furthermore, for decimal instructions, data in source fields may be accessed more than once, and intermediate values may be placed in the result field that may differ from the original operand and final result values. (See "Storage-Operand Consistency" on page 5-98) Thus, in a multiprocessing configuration, an instruction such as ADD DECIMAL cannot be safely used to update a shared storage location when the possibility exists that another CPU may also be updating that location.

Other Instructions for Decimal Operands

In addition to the decimal instructions in this chapter, MOVE NUMERICS and MOVE ZONES are provided for operating on data of lengths up to 256 bytes in the zoned format. Two instructions are provided for converting data between the zoned and packed formats: PACK transforms zoned data of lengths up to 16 bytes into packed data, and UNPACK performs the reverse transformation. MOVE WITH OFFSET can operate on packed data of lengths up to 16 bytes. Two instructions are provided for conversion between the packed-decimal and signed-binary-integer formats. CONVERT TO BINARY converts packed decimal to binary, and CONVERT TO DECIMAL
converts binary to packed decimal; the length of
the packed decimal operand of these instructions
is eight bytes (15 digits and sign) for CONVERT
TO BINARY (CVB) and CONVERT TO DECIMAL
(CVD), and sixteen bytes (31 digits and sign) for
CONVERT TO BINARY (CVBG) and CONVERT
TO DECIMAL (CVDG). These seven instructions
are not considered to be decimal instructions and
are described in Chapter 7, “General Instructions.”
The editing instructions in this chapter may also
be used to change data from the packed to the
zoned format.

Decimal-Operand Data Exception

A decimal-operand data exception is recognized
when any of the following is true:

1. The sign or digit codes of operands in the
decimal instructions or in CONVERT TO
BINARY (described in Chapter 7, “General
Instructions”) are invalid.

2. The operand fields in ADD DECIMAL,
COMPARE DECIMAL, DIVIDE DECIMAL,
multiply DECIMAL, and SUBTRACT
DECIMAL overlap in a way other than with
coincident rightmost bytes; or operand fields in
ZERO AND ADD overlap, and the rightmost
byte of the second operand is to the right of
the rightmost byte of the first operand. On
some models, the improper overlap of oper-
ands for ZERO AND ADD is not recognized
as a decimal-operand data exception; instead,
the operation is performed as if the entire
second operand were fetched before any byte
of the result is stored.

3. The multiplicand in MULTIPLY DECIMAL has
an insufficient number of leftmost zeros.

A decimal-operand data exception causes the
operation to be suppressed, except that, for EDIT
and EDIT AND MARK, the operation may be sup-
pressed or terminated. In the case of EDIT and
EDIT AND MARK, an invalid sign code cannot
occur.

Instructions

The decimal instructions and their mnemonics,
formats, and operation codes are listed in
Figure 8-2 on page 8-5. The figure also indicates
when the condition code is set, the instruction
fields that designate access registers, and the
exceptional conditions in operand designations,
data, or results that cause a program interruption.

Note: In the detailed descriptions of the indi-
vidual instructions, the mnemonic and the sym-
bolic operand designation for the assembler lan-
guage are shown with each instruction. For ADD
DECIMAL, for example, AP is the mnemonic and
D1(L1,B1),D2(L2,B2) the operand designation.

Programming Note: The decimal instruction
TEST DECIMAL is available when the extended-
translation facility 2 is installed.
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD DECIMAL</td>
<td>AP</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>COMPARE DECIMAL</td>
<td>CP</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>DIVIDE DECIMAL</td>
<td>DP</td>
<td>SS</td>
<td>A SP</td>
</tr>
<tr>
<td>EDIT</td>
<td>ED</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>EDIT AND MARK</td>
<td>EDMK</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>MULTIPLY DECIMAL</td>
<td>MP</td>
<td>SS</td>
<td>A SP</td>
</tr>
<tr>
<td>SHIFT AND ROUND DECIMAL</td>
<td>SRP</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>SUBTRACT DECIMAL</td>
<td>SP</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>TEST DECIMAL</td>
<td>TP</td>
<td>RSL C E₂</td>
<td>A</td>
</tr>
<tr>
<td>ZERO AND ADD</td>
<td>ZAP</td>
<td>SS C</td>
<td>A</td>
</tr>
</tbody>
</table>

**Explanation:**

A Access exceptions for logical addresses.
B₁ B₁ field designates an access register in the access-register mode.
B₂ B₂ field designates an access register in the access-register mode.
C Condition code is set.
Dd Decimal-operand data exception.
DF Decimal-overflow exception.
DK Decimal-divide exception.
E₂ Extended-translation facility 2.
G₁ Instruction execution includes the implied use of general register 1.
RSL RSL instruction format.
SP Specification exception.
SS SS instruction format.
ST PER storage-alteration event.

Figure 8-2. Summary of Decimal Instructions

**ADD DECIMAL**

\[ \text{AP } D₁(L₁,B₁),D₂(L₂,B₂) \ [\text{SS}] \]

<table>
<thead>
<tr>
<th>'FA'</th>
<th>L₁</th>
<th>L₂</th>
<th>B₁</th>
<th>D₁</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The second operand is added to the first operand, and the resulting sum is placed at the first-operand location. The operands and result are in the packed format.

Addition is algebraic, taking into account the signs and all digits of both operands. All sign and digit codes are checked for validity.

If the first operand is too short to contain all leftmost nonzero digits of the sum, decimal overflow occurs. The operation is completed. The result is obtained by ignoring the overflow digits, and condition code 3 is set. If the decimal-overflow mask is one, a program interruption for decimal overflow occurs.

The sign of the sum is determined by the rules of algebra. In the absence of overflow, the sign of a zero result is made positive. If overflow occurs, a zero result is given either a positive or negative sign, as determined by what the sign of the correct sum would have been.

**Resulting Condition Code:**

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

**Program Exceptions:**

- Access (fetch, operand 2; fetch and store, operand 1)
- Data
- Decimal overflow

**Programming Note:** An example of the use of the ADD DECIMAL instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."
The first operand is compared with the second operand, and the result is indicated in the condition code. The operands are in the packed format.

Comparison is algebraic and follows the procedure for decimal subtraction, except that both operands remain unchanged. When the difference is zero, the operands are equal. When a nonzero difference is positive or negative, the first operand is high or low, respectively.

Overflow cannot occur because the difference is discarded.

All sign and digit codes are checked for validity.

**Resulting Condition Code:**

0 Operands equal
1 First operand low
2 First operand high
3 --

**Program Exceptions:**

- Access (fetch, operands 1 and 2)
- Data

**Programming Notes:**

1. An example of the use of the COMPARE DECIMAL instruction is given in Appendix A, "Number Representation and Instruction-Use Examples."

2. The preferred and alternate sign codes for a particular sign are treated as equivalent for comparison purposes.

3. A negative zero and a positive zero compare equal.

The first operand (the dividend) is divided by the second operand (the divisor). The resulting quotient and remainder are placed at the first-operand location. The operands and results are in the packed format.

The quotient is placed leftmost in the first-operand location. The number of bytes in the quotient field is equal to the difference between the dividend and divisor lengths (L₁ - L₂). The remainder is placed rightmost in the first-operand location and has a length equal to the divisor length. Together, the quotient and remainder fields occupy the entire first operand; therefore, the address of the quotient is the address of the first operand.

The divisor length cannot exceed 15 digits and sign (L₂ not greater than seven) and must be less than the dividend length (L₂ less than L₁); otherwise, a specification exception is recognized.

The dividend, divisor, quotient, and remainder are each signed decimal integers in the packed format and are right-aligned in their fields. All sign and digit codes of the dividend and divisor are checked for validity.

The sign of the quotient is determined by the rules of algebra from the dividend and divisor signs. The sign of the remainder has the same value as the dividend sign. These rules hold even when the quotient or remainder is zero.

Overflow cannot occur. If the divisor is zero or the quotient is too large to be represented by the number of digits specified, a decimal-divide exception is recognized. This includes the case of division of zero by zero. The decimal-divide exception is indicated only if the sign codes of both the dividend and divisor are valid, and only if the digit or digits used in establishing the exception are valid.

**Condition Code:** The code remains unchanged.
Program Exceptions:

- Access (fetch, operand 2; fetch and store, operand 1)
- Data
- Decimal divide
- Specification

Programming Notes:

1. An example of the use of the DIVIDE DECIMAL instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. The dividend cannot exceed 31 digits and sign. Since the remainder cannot be shorter than one digit and sign, the quotient cannot exceed 29 digits and sign.

3. The condition for a decimal-divide exception can be determined by a trial comparison. The leftmost digit of the divisor is aligned one digit to the right of the leftmost dividend digit, with rightmost zeros appended up to the length of the dividend. When the divisor, so aligned, is less than or equal to the dividend, ignoring signs, a divide exception is indicated.

4. If a data exception does not exist, a decimal-divide exception occurs when the leftmost dividend digit is not zero.

EDIT

```
ED D1(L,B1),D2(B2) [SS]
```

The second operand (the source), which normally contains one or more decimal numbers in the packed format, is changed to the zoned format and modified under the control of the first operand (the pattern). The edited result replaces the first operand.

The length field specifies the length of the first operand, which may contain bytes of any value.

The length of the source is determined by the operation according to the contents of the pattern. The source normally consists of one or more decimal numbers, each in the packed format. The leftmost four bits of each source byte must specify a decimal-digit code (0000-1001); a sign code (1010-1111) is recognized as a data exception. The rightmost four bits may specify either a sign code or a decimal-digit code. Access and data exceptions are recognized only for those bytes in the second operand which are actually required.

The result is obtained as if both operands were processed left to right one byte at a time. Overlapping pattern and source fields give unpredictable results.

During the editing process, each byte of the pattern is affected in one of three ways:

1. It is left unchanged.
2. It is replaced by a source digit expanded to the zoned format.
3. It is replaced by the first byte in the pattern, called the fill byte.

Which of the three actions takes place is determined by one or more of the following: the type of the pattern byte, the state of the significance indicator, and whether the source digit examined is zero.

Pattern Bytes: There are four types of pattern bytes: digit selector, significance starter, field separator, and message byte. Their coding is as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Code (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digit selector</td>
<td>0010 0000</td>
</tr>
<tr>
<td>Significance starter</td>
<td>0010 0001</td>
</tr>
<tr>
<td>Field separator</td>
<td>0010 0010</td>
</tr>
<tr>
<td>Message byte</td>
<td>Any other</td>
</tr>
</tbody>
</table>

The detection of either a digit selector or a significance starter in the pattern causes an examination to be made of the significance indicator and of a source digit. As a result, either the expanded source digit or the fill byte, as appropriate, is selected to replace the pattern byte. Additionally, encountering a digit selector or a significance starter may cause the significance indicator to be changed.

The field separator identifies individual fields in a multiple-field editing operation. It is always replaced in the result by the fill byte, and the significance indicator is always off after the field separator is encountered.
Message bytes in the pattern are either replaced by the fill byte or remain unchanged in the result, depending on the state of the significance indicator. They may thus be used for padding, punctuation, or text in the significant portion of a field or for the insertion of sign-dependent symbols.

**Fill Byte:** The first byte of the pattern is used as the fill byte. The fill byte can have any code and may concurrently specify a control function. If this byte is a digit selector or significance starter, the indicated editing action is taken after the code has been assigned to the fill byte.

**Source Digits:** Each time a digit selector or significance starter is encountered in the pattern, a new source digit is examined for placement in the pattern field. Either the source digit is disregarded, or it is expanded to the zoned format, by appending the zone code 1111 on the left, and stored in place of the pattern byte.

Execution is as if the source digits were selected one byte at a time and as if a source byte were fetched for inspection only once during an editing operation. Each source digit is examined only once for a zero value. The leftmost four bits of each byte are examined first, and the rightmost four bits, when they represent a decimal-digit code, remain available for the next pattern byte that calls for a digit examination. When the leftmost four bits contain an invalid digit code, a data exception is recognized, and the operation is terminated.

At the time the left digit of a source byte is examined, the rightmost four bits are checked for the existence of a sign code. When a sign code is encountered in the rightmost four bit positions, these bits are not treated as a decimal-digit code, and a new source byte is fetched from storage when the next pattern byte calls for a source-digit examination.

When the pattern contains no digit selector or significance starter, no source bytes are fetched and examined.

**Significance Indicator:** The significance indicator is turned on or off to indicate the significance or nonsignificance, respectively, of subsequent source digits or message bytes. Significant source digits replace their corresponding digit selectors or significance starters in the result. Significant message bytes remain unchanged in the result.

The significance indicator, by its on or off state, indicates also the negative or positive value, respectively, of a completed source field and is used as one factor in the setting of the condition code.

The significance indicator is set to off at the start of the editing operation, after a field separator is encountered, or after a source byte is examined that has a plus code in the rightmost four bit positions.

The significance indicator is set to on when a significance starter is encountered whose source digit is a valid decimal digit, or when a digit selector is encountered whose source digit is a nonzero decimal digit, provided that in both instances the source byte does not have a plus code in the rightmost four bit positions.

In all other situations, the significance indicator is not changed. A minus sign code has no effect on the significance indicator.

**Result Bytes:** The result of an editing operation replaces and is equal in length to the pattern. It is composed of pattern bytes, fill bytes, and zoned source digits.

If the pattern byte is a message byte and the significance indicator is on, the message byte remains unchanged in the result. If the pattern byte is a field separator or if the significance indicator is off when a message byte is encountered in the pattern, the fill byte replaces the pattern byte in the result.

If the digit selector or significance starter is encountered in the pattern with the significance indicator off and the source digit zero, the source digit is considered nonsignificant, and the fill byte replaces the pattern byte. If the digit selector or significance starter is encountered with either the significance indicator on or with a nonzero decimal source digit, the source digit is considered significant, is changed to the zoned format, and replaces the pattern byte in the result.

**Condition Code:** The sign and magnitude of the last field edited are used to set the condition code. The term “last field” refers to those source digits, if
any, in the second operand selected by digit selectors or significance starters after the last field separator; if the pattern contains no field separator, there is only one field, which is considered to be the last field. If no such source digits are selected, the last field is considered to be of zero length.

Condition code 0 is set when the last field edited is zero or of zero length.

Condition code 1 is set when the last field edited is nonzero and the significance indicator is on. (This indicates a result less than zero if the last source byte examined contained a sign code in the rightmost four bits.)

Condition code 2 is set when the last field edited is nonzero and the significance indicator is off. (This indicates a result greater than zero if the last source byte examined contained a sign code in the rightmost four bits.)

Figure 8-3 on page 8-10 summarizes the functions of the EDIT and EDIT AND MARK operations. The leftmost four columns list all the significant combinations of the four conditions that can be encountered in the execution of an editing operation. The rightmost two columns list the action taken for each case — the type of byte placed in the result field and the new setting of the significance indicator.

**Resulting Condition Code:**

0  Last field zero or zero length
1  Last field less than zero
2  Last field greater than zero
3  --

**Program Exceptions:**

- Access (fetch, operand 2; fetch and store, operand 1)
- Data

**Programming Notes:**

1. Examples of the use of the EDIT instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”
2. Editing includes sign and punctuation control, and the suppression and protection of leading zeros by replacing them with blanks or asterisks. It also facilitates programmed blanking of all-zero fields. Several fields may be edited in one operation, and numeric information may be combined with text.
3. In most cases, the source is shorter than the pattern because each four-bit source digit produces an eight-bit byte in the result.
4. The total number of digit selectors and significance starters in the pattern always equals the number of source digits edited.
5. If the fill byte is a blank, if no significance starter exists in the pattern, and if the source digit examined for each digit selector is zero, the editing operation blanks the result field.
6. The resulting condition code indicates whether or not the last field is all zeros and, if nonzero, reflects the state of the significance indicator. The significance indicator reflects the sign of the source field only if the last source byte examined contains a sign code in the rightmost four bits. For multiple-field editing operations, the condition code reflects the sign and value only of the field following the last field separator.
7. Significant performance degradation is possible when, with DAT on, the second-operand address of an EDIT instruction designates a location that is closer to the left of a 4K-byte boundary than the length of the first operand of that instruction. This is because the machine may perform a trial execution of the instruction to determine if the second operand actually crosses the boundary. The second operand of EDIT, while normally shorter than the first operand, can in the extreme case have the same length as the first.

**EDIT AND MARK**

```
EDMK D1(L,B1),D2(B2) [SS]
```

The second operand (the source), which normally contains one or more decimal numbers in the packed format, is changed to the zoned format and modified under the control of the first operand (the pattern). The address of the first significant result byte is inserted in general register 1. The edited result replaces the pattern.
<table>
<thead>
<tr>
<th>Pattern Byte</th>
<th>Previous State of Significance Indicator</th>
<th>Source Digit</th>
<th>Right Four Source Bits Are Plus Code</th>
<th>Result Byte</th>
<th>State of Significance Indicator at End of Digit Examination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digit selector</td>
<td>Off</td>
<td>0</td>
<td>*</td>
<td>Fill byte</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>1-9</td>
<td>No</td>
<td>Source digit #</td>
<td>Source digit</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td>On</td>
<td>0-9</td>
<td>No</td>
<td>Source digit</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>On</td>
<td>0-9</td>
<td>Yes</td>
<td>Source digit</td>
<td>Off</td>
</tr>
<tr>
<td>Significance starter</td>
<td>Off</td>
<td>0</td>
<td>No</td>
<td>Fill byte</td>
<td>On</td>
</tr>
<tr>
<td></td>
<td>1-9</td>
<td>Yes</td>
<td>Source digit #</td>
<td>Source digit</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>On</td>
<td>0-9</td>
<td>No</td>
<td>Source digit</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>On</td>
<td>0-9</td>
<td>Yes</td>
<td>Source digit</td>
<td>Off</td>
</tr>
<tr>
<td>Field separator</td>
<td>*</td>
<td>**</td>
<td>**</td>
<td>Fill byte</td>
<td>Off</td>
</tr>
<tr>
<td>Message byte</td>
<td>Off</td>
<td>**</td>
<td>**</td>
<td>Fill byte</td>
<td>Off</td>
</tr>
<tr>
<td></td>
<td>On</td>
<td>**</td>
<td>**</td>
<td>Message byte</td>
<td>On</td>
</tr>
</tbody>
</table>

**Explanation:**

* No effect on result byte or on new state of significance indicator.
** Not applicable because source is not examined.
# For EDIT AND MARK only, the address of the rightmost such result byte is placed in general register 1.

See Figure 8-3 for a summary of the EDIT and EDIT AND MARK operations.

**Resulting Condition Code:**

- 0 Last field zero or zero length
- 1 Last field less than zero
- 2 Last field greater than zero
- 3 --

**Program Exceptions:**

- Access (fetch, operand 2; fetch and store, operand 1)
- Data

**Programming Notes:**

1. Examples of the use of the EDIT AND MARK instruction are given in Appendix A, "Number Representation and Instruction-Use Examples."

2. EDIT AND MARK facilitates the programming of floating currency-symbol insertion. Using
appropriate source and pattern data, the address inserted in general register 1 is one greater than the address where a floating currency-sign would be inserted. BRANCH ON COUNT (BCTR, BCTGR), with zero in the R field, may be used to reduce the inserted address by one.

3. No address is inserted in general register 1 when the significance indicator is turned on as a result of encountering a significance starter with the corresponding source digit zero. To ensure that general register 1 contains a proper address when this occurs, the address of the pattern byte that immediately follows the appropriate significance starter could be placed in the register beforehand.

4. When multiple fields are edited with one execution of the EDIT AND MARK instruction, the address, if any, inserted in general register 1 applies to the rightmost field edited for which the criteria were met.

5. See also the programming note under EDIT regarding performance degradation due to a possible trial execution.

MULTIPLY DECIMAL

MP D1(L1,B1),D2(L2,B2) [SS]

\[ 'FC' \quad L_1 \quad L_2 \quad B_1 \quad D_1 \quad B_2 \quad D_2 \]

0 8 12 16 20 32 36 47

The product of the first operand (the multiplicand) and the second operand (the multiplier) is placed at the first-operand location. The operands and result are in the packed format.

The multiplier length cannot exceed 15 digits and sign (L not greater than seven) and must be less than the multiplicand length (L less than L1); otherwise, a specification exception is recognized.

The multiplicand must have at least as many bytes of leftmost zeros as the number of bytes in the multiplier; otherwise, a data exception is recognized. This restriction ensures that no product overflow occurs.

The multiplicand, multiplier, and product are each signed decimal integers in the packed format and are right-aligned in their fields. All sign and digit codes of the multiplicand and multiplier are checked for validity. The sign of the product is determined by the rules of algebra from the multiplier and multiplicand signs, even if one or both operands are zeros.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2; fetch and store, operand 1)
- Data
- Specification

**Programming Notes:**
1. An example of the use of the MULTIPLY DECIMAL instruction is given Appendix A, “Number Representation and Instruction-Use Examples.”
2. The product cannot exceed 31 digits and sign. The leftmost digit of the product is always zero.

SHIFT AND ROUND DECIMAL

SRP D1(L1,B1),D2(B2),I3 [SS]

\[ 'FO' \quad L_1 \quad I_3 \quad B_1 \quad D_1 \quad B_2 \quad D_2 \]

0 8 12 16 20 32 36 47

The first operand is shifted in the direction and for the number of decimal-digit positions specified by the second-operand address, and, when shifting to the right is specified, the absolute value of the first operand is rounded by the rounding digit, I3. The first operand and the result are in the packed format.

The first operand is considered to be in the packed-decimal format. Only its digit portion is shifted; the sign position does not participate in the shifting. Zeros are supplied for the vacated digit positions. The result replaces the first operand. Nothing is stored outside of the specified first-operand location.

The second-operand address, specified by the B2 and D2 fields, is not used to address data; bits 58-63 of that address are the shift value, and the leftmost bits of the address are ignored.
The shift value is a six-bit signed binary integer, indicating the direction and the number of decimal-digit positions to be shifted. Positive shift values specify shifting to the left. Negative shift values, which are represented in two’s complement notation, specify shifting to the right. The following are examples of the interpretation of shift values:

<table>
<thead>
<tr>
<th>Shift Value</th>
<th>Amount and Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>011111</td>
<td>31 digits to the left</td>
</tr>
<tr>
<td>000001</td>
<td>One digit to the left</td>
</tr>
<tr>
<td>000000</td>
<td>No shift</td>
</tr>
<tr>
<td>111111</td>
<td>One digit to the right</td>
</tr>
<tr>
<td>100000</td>
<td>32 digits to the right</td>
</tr>
</tbody>
</table>

For a right shift, the I3 field, bits 12-15 of the instruction, is used as a decimal rounding digit. The first operand, which is treated as positive by ignoring the sign, is rounded by decimally adding the rounding digit to the leftmost of the digits to be shifted out and by propagating the carry, if any, to the left. The result of this addition is then shifted right. Except for validity checking and the participation in rounding, the digits shifted out of the rightmost decimal-digit position are ignored and are lost.

If one or more nonzero digits are shifted out during a left shift, decimal overflow occurs. The operation is completed. The result is obtained by ignoring the overflow digits, and condition code 3 is set. If the decimal-overflow mask is one, a program interruption for decimal overflow occurs. Overflow cannot occur for a right shift, with or without rounding, or when no shifting is specified.

In the absence of overflow, the sign of a zero result is made positive. If overflow occurs, the sign of the result is the same as the original sign but with the preferred sign code.

A data exception is recognized when the first operand does not have valid sign and digit codes or when the rounding digit is not a valid digit code. The validity of the first-operand codes is checked even when no shift is specified, and the validity of the rounding digit is checked even when no addition for rounding takes place.

**Resulting Condition Code:**

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

**Program Exceptions:**

- Access (fetch and store, operand 1)
- Data
- Decimal overflow

**Programming Notes:**

1. Examples of the use of the SHIFT AND ROUND DECIMAL instruction are given in Appendix A, “Number Representation and Instruction-Use Examples.”

2. SHIFT AND ROUND DECIMAL can be used for shifting up to 31 digit positions left and up to 32 digit positions right. This is sufficient to clear all digits of any decimal number even with rounding.

3. For right shifts, the rounding digit 5 provides conventional rounding of the result. The rounding digit 0 specifies truncation without rounding.

4. When the B2 field is zero, the six-bit shift value is obtained directly from bits 42-47 of the instruction.

**SUBTRACT DECIMAL**

```
SP  D1(L1,B1),D2(L2,B2)  [SS]

<table>
<thead>
<tr>
<th>'FB'</th>
<th>L1</th>
<th>L2</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>
```

The second operand is subtracted from the first operand, and the resulting difference is placed at the first-operand location. The operands and result are in the packed format.

SUBTRACT DECIMAL is executed the same as ADD DECIMAL, except that the second operand is considered to have a sign opposite to the sign in storage. The second operand in storage remains unchanged.

**Resulting Condition Code:**

0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow
Program Exceptions:
- Access (fetch, operand 2; fetch and store, operand 1)
- Data
- Decimal overflow

TEST DECIMAL
TP D1(L1,B1) [RSL]

The first operand is tested for valid decimal digits and a valid sign code, and the result is indicated in the condition code. The operand is in the packed format.

Resulting Condition Code:
0 All digit codes and the sign valid
1 Sign invalid
2 At least one digit code invalid
3 Sign invalid and at least one digit code invalid

Program Exceptions:
- Access (fetch, operand 1)
- Operation (if the extended-translation facility 2 is not installed)

ZERO AND ADD
ZAP D1(L1,B1),D2(L2,B2) [SS]

The second operand is placed at the first-operand location. The operation is equivalent to an addition to zero. The operand and result are in the packed format.

Only the second operand is checked for valid sign and digit codes. Extra zeros are supplied on the left for the shorter operand if needed.

If the first operand is too short to contain all leftmost nonzero digits of the second operand, decimal overflow occurs. The operation is completed. The result is obtained by ignoring the overflow digits, and condition code 3 is set. If the decimal-overflow mask is one, a program interruption for decimal overflow occurs.

In the absence of overflow, the sign of a zero result is made positive. If overflow occurs, a zero result is given the sign of the second operand but with the preferred sign code.

The two operands may overlap, provided the rightmost byte of the first operand is coincident with or to the right of the rightmost byte of the second operand. In this case, the result is obtained as if the operands were processed right to left. When the operands overlap and the rightmost byte of the first operand is to the left of the rightmost byte of the second operand, then, depending on the model, either a data exception is recognized or the result is obtained as if the entire second operand were fetched before any byte of the result is stored.

Resulting Condition Code:
0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

Program Exceptions:
- Access (fetch, operand 2; store, operand 1)
- Data
- Decimal overflow

Programming Note: An example of the use of the ZERO AND ADD instruction is given in Appendix A, “Number Representation and Instruction-Use Examples.”
Chapter 9. Floating-Point Overview and Support Instructions

Registers And Controls .................................. 9-2
  Floating-Point Registers ................................ 9-2
    Additional Floating-Point (AFP) Registers .......... 9-2
    Valid Floating-Point-Register Designations ........ 9-2
    Floating-Point-Control (FPC) Register ............... 9-2
    AFP-Register-Control Bit ................................ 9-2
  Explicit Rounding Methods ............................. 9-3
    Summary of Rounding Action ........................... 9-3
Comparison of BFP and HFP Number Representations .... 9-4
BFP and HFP Number Ranges ............................... 9-4
  Equivalent BFP and HFP Number Representations .... 9-4
Instructions .............................................. 9-6
  CONVERT BFP TO HFP .................................. 9-8
  CONVERT HFP TO BFP .................................. 9-9
  LOAD .................................................... 9-10
  LOAD ZERO ............................................ 9-11
  STORE ................................................... 9-11
  Summary of All Floating-Point Instructions ........... 9-12

Floating-point instructions are used to perform calculations on operands having a wide range of magnitude and to obtain results scaled to preserve precision.

Floating-point operands have formats based on either the radix 16 or the radix 2. The radix values 16 and 2 lead to the terminology “hexadecimal” and “binary” floating point (HFP and BFP). The formats are also based on three operand lengths: short (32 bits), long (64 bits), and extended (128 bits). Short operands require less storage than long or extended operands. On the other hand, long and extended operands permit greater precision in computation.

A floating-point operand may be numeric or, for BFP only, nonnumeric (a not-a-number, or NaN). A numeric operand, called a floating-point number, has three components: a sign bit, a signed binary exponent, and a significand. The significand consists of an implicit unit digit to the left of an implied radix point and an explicit fraction field to the right. The significand digits are based on the radix, 2 or 16. The magnitude (an unsigned value) of the number is the product of the significand and the radix raised to the power of the exponent. The number is positive or negative depending on whether the sign bit is zero or one, respectively. A nonnumeric BFP operand also has a sign bit, signed exponent, and fraction field.

Hexadecimal-floating-point (HFP) operands have formats which provide for exponents that specify powers of the radix 16 and significands that are hexadecimal numbers. The exponent range is the same for the short, long, and extended formats. The results of most operations on HFP data are truncated to fit into the target format, but there are instructions available to round the result when converting to a narrower format. For HFP operands, the implicit unit digit of the significand is always zero. Since the value of the significand and fraction are the same, HFP operations are described in terms of the fraction, and the term significand is not used.

Binary-floating-point (BFP) operands have formats which provide for exponents that specify powers of the radix 2 and significands that are binary numbers. The exponent range differs for different formats, the range being greater for the longer formats. In the long and extended formats, the exponent range is significantly greater for BFP data than for HFP data. The results of operations performed on BFP data are rounded automatically to fit into the target format; the manner of rounding is determined by a program-settable rounding mode.

Either normalized or unnormalized numbers may be used as operands for any HFP operation, where a normalized number is one having a nonzero leftmost fraction digit. Most HFP instructions generate normalized results for greatest precision. HFP add and subtract instructions that generate unnormalized results are also available.

There are no unnormalized operands for BFP operations. For normalized BFP numbers, the implicit unit digit of the significand is one. For
values too small in magnitude to be represented in normalized form, the implicit unit digit is zero. These numbers are called “denormalized” numbers. Unlike the HFP format, where the same value can have multiple representations in a given format because of the possibility of unnormalized numbers, the BFP format does not allow such redundancy.

Both BFP and HFP data formats appear in storage in the same left-to-right sequence as all other data formats. Bits of a data format that are numbered 0-7 constitute the byte in the leftmost (lowest-numbered) byte location in storage, bits 8-15 form the byte in the next sequential location, and so on. (See also the section “Storage Addressing” on page 3-2)

Most of the floating-point instructions are defined in detail in this publication in Chapter 18, “Hexadecimal-Floating-Point Instructions,” and Chapter 19, “Binary-Floating-Point Instructions.” This chapter, Chapter 9, defines in detail instructions called floating-point-support (FPS) instructions. The FPS instructions either have operands that may be in either the BFP or the HFP format or have the function of converting between the two formats. This chapter also provides summary information about all of the floating-point instructions.

---

### Registers And Controls

#### Floating-Point Registers

All floating-point instructions (FPS, BFP, and HFP) use the same 16 floating-point registers. The floating-point registers are identified by the numbers 0-15 and are designated by a four-bit R field in floating-point instructions. Each floating-point register is 64 bits long and can contain either a short (32-bit) or a long (64-bit) floating-point operand.

A short floating-point number requires only the leftmost 32 bit positions of a floating-point register. The rightmost 32 bit positions of the register are ignored when the register is the source of an operand in the short format, and they remain unchanged when a short result is placed in the register.

A number in the extended (128-bit) format occupies a register pair. Register pairs are formed by coupling the 16 registers as follows: 0 and 2, 4 and 6, 8 and 10, 12 and 14, 1 and 3, 5 and 7, 9 and 11, and 13 and 15.

Each of the eight pairs is referred to by the number of the lower-numbered register of the pair.

#### Additional Floating-Point (AFP) Registers

Floating-point registers 0, 2, 4, and 6 are ones that were originally available on ESA/390 models. The remaining 12 floating-point registers (1, 3, 5, and 7-15) were added to ESA/390 and are referred to as the additional floating-point (AFP) registers. The AFP registers can be used only if bit 45 of control register 0, the AFP-register-control bit, is one. Attempting to use an AFP register when the AFP-register-control bit is zero results in an AFP-register data exception (DXC 1).

#### Valid Floating-Point-Register Designations

Any installed register may be designated by an instruction to specify the register location of a short or long floating-point operand.

An instruction specifying a floating-point operand in the extended format must designate register 0, 1, 4, 5, 8, 9, 12, or 13; otherwise, a specification exception is recognized.

#### Floating-Point-Control (FPC) Register

The floating-point-control (FPC) register is a 32-bit register that contains mask bits, flag bits, a data-exception code, and rounding-mode bits. The FPC register is described in the section “Floating-Point-Control (FPC) Register” on page 19-2.

#### AFP-Register-Control Bit

Bit 45 of control register 0 is the AFP-register-control bit. The AFP registers and the BFP instructions can be used successfully only when the AFP-register-control bit is one. Attempting to use one of the 12 additional floating-point registers when the AFP-register-control bit is zero results in an AFP-register data exception (DXC 1). Attempting to execute any BFP instruc-
tion when the AFP-register-control bit is zero results in a BFP-instruction data exception (DXC 2). If the conditions for both DXC 1 and DXC 2 exist, DXC 1 is reported. If the conditions for both a data exception and a specification exception exist, it is unpredictable which exception is reported.

The initial value of the AFP-register-control bit is zero.

Explicit Rounding Methods

The floating-point-support instruction CONVERT HFP TO BFP includes an \( M^3 \) modifier field which can specify any of five rounding methods. One HFP instruction (CONVERT TO FIXED) and three BFP instructions (CONVERT TO FIXED, DIVIDE TO INTEGER, and LOAD FP INTEGER) also include either an \( M^3 \) modifier field or a similar \( M^4 \) modifier field. The five rounding methods are as follows:

\[ \text{M}^3 \text{ or M}^4 \]

**Rounding Method**

1. **Biased round to nearest:**
   Round the intermediate result up or down to the nearest representable value; that is, add, ignoring the sign, a one to the bit just beyond the last result bit to be retained, propagate the carry, and discard the bits beyond the last one to be retained.

4. **Round to nearest:**
   Round the intermediate result up or down to the nearest representable value; that is, add, ignoring the sign, a one to the bit just beyond the last result bit to be retained, propagate the carry, and discard the bits beyond the last one to be retained. If the difference was exactly one-half ulp (unit in the last place — a one in the bit position just beyond the last place, with all zeros beyond that), the nearest even number is chosen; that is, after the rounding addition, the last result bit retained is set to zero.

5. **Round toward 0:**
   Discard all bits to the right of the last intermediate-result bit to be retained.

6. **Round toward +\( \infty \):**
   If the intermediate result is positive and there are any ones to the right of the last result bit to be retained, add one to that bit. Then, for either sign, discard the bits beyond the last one to be retained.

7. **Round toward -\( \infty \):**
   If the intermediate result is negative and there are any ones to the right of the last result bit to be retained, subtract one from that bit (that is, add one to the magnitude). Then, for either sign, discard the bits beyond the last one to be retained.

The handling of an \( M^3 \) or \( M^4 \) value of zero depends on the type of instruction. For BFP instructions, an \( M^3 \) or \( M^4 \) value of zero causes rounding to be performed according to the current rounding mode specified in the FPC register. The floating-point-support and HFP instructions treat an \( M^3 \) or \( M^4 \) of zero the same as 5, that is, round toward zero.

**Summary of Rounding Action**

[Figure 9-1 on page 9-4] summarizes the rounding action for floating-point-support (FPS), BFP, and HFP instructions.
### Comparison of BFP and HFP Number Representations

#### BFP and HFP Number Ranges

Figure 9-2 shows the range of numbers, in decimal form, that can be represented in different floating-point formats.

<table>
<thead>
<tr>
<th>Type</th>
<th>Short</th>
<th>Long</th>
<th>Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nmax</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFP</td>
<td>$\pm3.4\times10^{38}$</td>
<td>$\pm1.8\times10^{308}$</td>
<td>$\pm1.2\times10^{4932}$</td>
</tr>
<tr>
<td>HFP</td>
<td>$\pm7.2\times10^{75}$</td>
<td>$\pm7.2\times10^{75}$</td>
<td>$\pm7.2\times10^{75}$</td>
</tr>
<tr>
<td><strong>Nmin</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFP</td>
<td>$\pm1.2\times10^{-38}$</td>
<td>$\pm2.2\times10^{-308}$</td>
<td>$\pm3.4\times10^{-4932}$</td>
</tr>
<tr>
<td>HFP</td>
<td>$\pm5.5\times10^{-79}$</td>
<td>$\pm5.5\times10^{-79}$</td>
<td>$\pm5.5\times10^{-79}$</td>
</tr>
<tr>
<td><strong>Dmin</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFP</td>
<td>$\pm1.4\times10^{-45}$</td>
<td>$\pm4.9\times10^{-324}$</td>
<td>$\pm6.5\times10^{-4966}$</td>
</tr>
<tr>
<td>HFP</td>
<td>$\pm5.2\times10^{-85}$</td>
<td>$\pm1.2\times10^{-94}$</td>
<td>$\pm1.7\times10^{-111}$</td>
</tr>
</tbody>
</table>

**Explanation:**
- **Dmin** Smallest (in magnitude) representable denormalized (BFP) or nonzero unnormalized (HFP) number.
- **Nmax** Largest (in magnitude) representable number.
- **Nmin** Smallest (in magnitude) representable normalized number.
- Values are decimal approximations.

#### Equivalent BFP and HFP Number Representations

The exponent of an HFP number is represented in the number as an unsigned seven-bit binary integer called the characteristic. The characteristic is obtained by adding 64 to the exponent value (excess-64 notation). The range of the characteristic is 0 to 127, which corresponds to an exponent range of $-64$ to $+63$.

The exponent of a BFP number is represented in the number as an unsigned binary integer called the biased exponent. The biased exponent is obtained by adding a bias to the exponent value. The number of bit positions containing the biased exponent, the value of the bias, and the exponent range depend on the number format (short, long, or extended) and are shown for the three formats in Figure 19-7 on page 19-5.

Biased exponents
are similar to the characteristics of the HFP format, except that special meanings are attached to biased exponents of all zeros and all ones, which are discussed in the section "Classes of BFP Data" on page 19-5.

In each of the three BFP or HFP formats, the binary or hexadecimal point of a number, respectively, is considered to be to the left of the leftmost fraction digit. To the left of the point there is an implied unit digit, which is considered to be zero for HFP numbers or, for BFP numbers, one for normalized numbers and zero for zeros and denormalized numbers.

Figure 9-3 and Figure 9-4 on page 9-6 give examples of the closest representation of the same numbers in the BFP and HFP formats, with BFP values being rounded to nearest and HFP values being truncated.

The figures do not necessarily show the results of BFP/HFP conversions exactly. Rounding errors may make a small difference. Also, Figure 9-3 shows corresponding rounded short-format numbers, not the long HFP results of conversion from short BFP operands.

<table>
<thead>
<tr>
<th>Value</th>
<th>S BE or C Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>B 0 01111111 000000000000000000000000</td>
</tr>
<tr>
<td>0.5</td>
<td>B 0 01111110 000000000000000000000000</td>
</tr>
<tr>
<td>1/64</td>
<td>B 0 01111011 000000000000000000000000</td>
</tr>
<tr>
<td>0</td>
<td>B 0 00000000 000000000000000000000000</td>
</tr>
<tr>
<td>-0</td>
<td>B 1 00000000 000000000000000000000000</td>
</tr>
<tr>
<td>-15.0</td>
<td>B 1 10000010 111000000000000000000000</td>
</tr>
<tr>
<td>20/7</td>
<td>B 0 10000000 01101101101101101101110</td>
</tr>
<tr>
<td>2−126</td>
<td>B 0 00000001 000000000000000000000000</td>
</tr>
<tr>
<td>2−149</td>
<td>B 0 00000000 000000000000000000000001</td>
</tr>
<tr>
<td>2128xF</td>
<td>B 0 11111111 111111111111111111111111 F=1−2−24</td>
</tr>
<tr>
<td>2−260</td>
<td>B Zero (number too small)</td>
</tr>
<tr>
<td>2248xF</td>
<td>B Not representable</td>
</tr>
</tbody>
</table>

**Explanation:**

- **B** BFP.
- **BE or C** Biased exponent of BFP number or characteristic of HFP number.
- **H** HFP.
- **S** Sign.

Figure 9-3. Examples of FP and HFP Numbers in Short Format
Instructions

The floating-point-support instructions and their mnemonics and operation codes are listed in Figure 9-5 on page 9-7. The figure indicates, in the column labeled “Characteristics,” the instruction format, when the condition code is set, the instruction fields that designate access registers, and the exceptional conditions in operand designations, data, or results that cause a program interruption.

All floating-point-support instructions are subject to the AFP-register-control bit, bit 45 of control register 0. The AFP-register-control bit must be one when an AFP register is specified as an operand location; otherwise, an AFP-register data exception, DXC 1, is recognized.

Mnemonics for the floating-point instructions have an R as the last letter when the instruction is in the RR, RRE, or RRF format. Certain letters are used for floating-point instructions to represent operand-format length, as follows:

- D Long
- E Short
- X Extended

Note: In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designation for the assembler lan-
language are shown with each instruction. For a register-to-register operation using LOAD (short), for example, LER is the mnemonic and R₁,R₂ the operand designation.

Programming Note: The following additional floating-point-support instructions are available when the long-displacement facility is installed:
- LOAD (LDY, LEY)
- STORE (STDY, STEY)

<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVERT BFP TO HFP (long)</td>
<td>THDR</td>
<td>RRE C</td>
<td>Da</td>
</tr>
<tr>
<td>CONVERT BFP TO HFP (short to long)</td>
<td>THDER</td>
<td>RRE C</td>
<td>Da</td>
</tr>
<tr>
<td>CONVERT HFP TO BFP (long)</td>
<td>TBDR</td>
<td>RRF C</td>
<td>SP Da</td>
</tr>
<tr>
<td>CONVERT HFP TO BFP (long to short)</td>
<td>TBEDR</td>
<td>RRF C</td>
<td>SP Da</td>
</tr>
<tr>
<td>LOAD (extended)</td>
<td>LXR</td>
<td>RRE</td>
<td>SP Da</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>LDR</td>
<td>RRE</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>LD</td>
<td>RX A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>LDY</td>
<td>RXY LD A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD (short)</td>
<td>LER</td>
<td>RR A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD (short)</td>
<td>LE</td>
<td>RX A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD (short)</td>
<td>LEY</td>
<td>RXY LD A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD ZERO (extended)</td>
<td>LZXR</td>
<td>RRE SP Da</td>
<td></td>
</tr>
<tr>
<td>LOAD ZERO (long)</td>
<td>LZXR</td>
<td>RRE Da</td>
<td></td>
</tr>
<tr>
<td>LOAD ZERO (short)</td>
<td>LZER</td>
<td>RRE Da</td>
<td></td>
</tr>
<tr>
<td>STORE (long)</td>
<td>STD</td>
<td>RX A</td>
<td>Da</td>
</tr>
<tr>
<td>STORE (long)</td>
<td>STDY</td>
<td>RXY LD A</td>
<td>Da</td>
</tr>
<tr>
<td>STORE (short)</td>
<td>STE</td>
<td>RX A</td>
<td>Da</td>
</tr>
<tr>
<td>STORE (short)</td>
<td>STEY</td>
<td>RXY LD A</td>
<td>Da</td>
</tr>
</tbody>
</table>

Explanation:

A    Access exceptions for logical addresses.
B2   B2 field designates an access register in the access-register mode.
C    Condition code is set.
Da   AFP-register data exception.
LD   Long-displacement facility.
RR   RR instruction format.
RRE  RRE instruction format.
RF   RRF instruction format.
RX   RX instruction format.
SP   Specification exception.
ST   PER storage-alteration event.

Figure 9-5. Summary of Floating-Point-Support Instructions
CONVERT BFP TO HFP

Mnemonic R₁,R₂ [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>// ///</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
THDER 'B358' Short BFP operand, long HFP result
THDR 'B359' Long BFP operand, long HFP result

The second operand (the source operand) is converted from the binary-floating-point (BFP) format to the hexadecimal-floating-point (HFP) format, and the normalized result is placed at the first-operand location. The sign and magnitude of the source operand are tested to determine the setting of the condition code.

For numeric operands, the sign of the result is the sign of the source operand. If the source operand has a sign bit of one and all other operand bits are zeros, the result also is a one followed by all zeros.

When, for THDR, the characteristic of the result would be negative, the result is made all zeros but with the same sign as that of the source operand, and condition code 1 or 2 is set to indicate the sign of the source operand.

When, for THDR, the characteristic of the hexadecimal intermediate result is too large to fit into the target format, the result is set to all ones (that is, the largest-in-magnitude representable number) but with the same sign as that of the source operand, and condition code 3 is set.

See Figure 9-6 for a detailed description of the results of this instruction.

**Resulting Condition Code:**

0  Source was zero
1  Source was less than zero
2  Source was greater than zero
3  Special case

**Program Exceptions:**

- Data with DXC 1, AFP register

**Programming Notes:**

1. The BFP-to-HFP conversion instructions are summarized in Figure 9-7 on page 9-9.

2. CONVERT BFP TO HFP (THDER) converts BFP operands in the short format to HFP operands in the long format, rather than converting short to short, to retain full precision. Using this long HFP result subsequently as a short operand requires no extra conversion steps.

<table>
<thead>
<tr>
<th>Source Operand (a)</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞ ≤ a &lt; −Hmax</td>
<td>T(−Hmax), cc3</td>
</tr>
<tr>
<td>−Hmax ≤ a ≤ −Hmin</td>
<td>T(r), cc1</td>
</tr>
<tr>
<td>−Hmin &lt; a &lt; 0</td>
<td>T(−0)¹, cc1</td>
</tr>
<tr>
<td>−0</td>
<td>T(−0), cc0</td>
</tr>
<tr>
<td>+0</td>
<td>T(+0), cc0</td>
</tr>
<tr>
<td>0 &lt; a &lt; +Hmin</td>
<td>T(+0)², cc2</td>
</tr>
<tr>
<td>+Hmin ≤ a ≤ +Hmax</td>
<td>T(r), cc2</td>
</tr>
<tr>
<td>+Hmax &lt; a ≤ +∞</td>
<td>T(+Hmax), cc3</td>
</tr>
<tr>
<td>NaN</td>
<td>T(+Hmax), cc3</td>
</tr>
</tbody>
</table>

**Explanation:**

1. Condition code 1 is set to indicate the source was less than zero.
2. Condition code 2 is set to indicate the source was greater than zero.
ccn  Condition code is set to n.
r  The value derived when the BFP source value a is converted to the HFP format. This result is always exact.
Hmax Largest (in magnitude) representable number in the target HFP format.
Hmin Smallest (in magnitude) representable normalized number in the target HFP format.
T(x)  The value x is placed at the target operand location.

Figure 9-6. Results: CONVERT BFP TO HFP
### CONVERT HFP TO BFP

**Mnemonic**  
R₁, M₃, R₂  

<table>
<thead>
<tr>
<th>Op Code</th>
<th>M₃</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

**Mnemonic**  
Op Code  

- **TBEDR**  
  'B350'  
  Long HFP operand, short BFP result  

- **TBDR**  
  'B351'  
  Long HFP operand, long BFP result  

The second operand (the source operand) is converted from the hexadecimal-floating-point (HFP) format to the binary-floating-point (BFP) format, and the result rounded according to the rounding method specified by the M₃ field is placed at the first-operand location. The sign and magnitude of the source operand are tested to determine the setting of the condition code.

The M₃ field contains a modifier specifying a rounding method, as follows:

- **M₃**  
  **Rounding Method**  
  0 Round toward 0  
  1 Biased round to nearest  
  4 Round to nearest  
  5 Round toward 0  
  6 Round toward +∞  
  7 Round toward −∞

A modifier other than 0, 1, or 4-7 is invalid.

The sign of the result is the sign of the second operand. If the second operand has a sign bit of one and all other operand bits are zeros, the result also is a one followed by all zeros.

See [Figure 9-8 on page 9-10](#) for a detailed description of the results of this instruction.

The M₃ field must designate a valid modifier; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

- 0 Source was zero  
- 1 Source was less than zero  
- 2 Source was greater than zero  
- 3 Special case

**Program Exceptions:**

- Data with DXC 1, AFP register  
- Specification

**Programming Notes:**

1. The HFP-to-BFP conversion instructions are summarized in Figure 9-7.

2. Conversion to short BFP numbers requires HFP operands in the long format; a short HFP operand should be extended to long by ensuring that the right half of the register is cleared. Thus, the entire register should be cleared before loading a short HFP operand into it for conversion to BFP. This avoids unrepeatable rounding errors in the BFP result due to data left over from previous use.
Source Operand (a) | Results
---|---
$a < -N_{\text{max}}$ | See Part 2 of this figure.
$-N_{\text{max}} \leq a \leq -N_{\text{min}}$ | $T(r), cc1$
$-N_{\text{min}} < a \leq -D_{\text{min}}$ | $T(d), cc1$
$-D_{\text{min}} < a < 0$ | $T(d)^1, cc1$
$0 < a < +D_{\text{min}}$ | $T(d)^2, cc2$
$+D_{\text{min}} \leq a < +N_{\text{min}}$ | $T(d), cc2$
$+N_{\text{min}} \leq a \leq +N_{\text{max}}$ | $T(r), cc2$
$+N_{\text{max}} < a$ | See Part 2 of this figure.

Figure 9-8 (Part 1 of 2). Results: CONVERT HFP to BFP

<table>
<thead>
<tr>
<th>Source Operand (a)</th>
<th>Results for Rounding Method Specified in $M_{3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Biased Round to Nearest</td>
</tr>
<tr>
<td>$a &lt; -N_{\text{max}}$</td>
<td>$T(-\infty), cc3$</td>
</tr>
<tr>
<td>$+N_{\text{max}} &lt; a$</td>
<td>$T(+\infty), cc3$</td>
</tr>
</tbody>
</table>

Explanation:
1. Condition code 1 is set for this case, even when the rounded result is zero.
2. Condition code 2 is set for this case, even when the rounded result is zero.
ccn Condition code is set to n.
d The denormalized value derived when the HFP source value $a$ is rounded to the format of the target using the rounding method specified in the $M_{3}$ field.
r The value derived when the HFP source value $a$ is rounded to the format of the target using the rounding method specified in the $M_{3}$ field.
$D_{\text{min}}$ Smallest (in magnitude) representable denormalized number in the target BFP format.
$N_{\text{max}}$ Largest (in magnitude) representable finite number in the target BFP format.
$N_{\text{min}}$ Smallest (in magnitude) representable normalized number in the target BFP format.
$T(x)$ The value $x$ is placed at the target operand location.

Figure 9-8 (Part 2 of 2). Results: CONVERT HFP to BFP

LOAD

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>R1,R2</th>
<th>[RR]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mnemonic2</td>
<td>R1,R2</td>
<td>[RRE]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mnemonic2</td>
<td>Op Code</td>
<td>Operands</td>
</tr>
<tr>
<td>LXR</td>
<td>'B365'</td>
<td>Extended</td>
</tr>
</tbody>
</table>

9-10 z/Architecture Principles of Operation
LOAD ZERO

Mnemonic  

\[
\begin{array}{cccccc}
\text{Op Code} & R_1 & X_2 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 31
\end{array}
\]

The second operand is placed unchanged at the first-operand location.

The operation is performed without inspecting the contents of the second operand; no arithmetic exceptions are recognized.

For LXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

The displacement for LE and LD is treated as a 12-bit unsigned binary integer. The displacement for LEY and LDY is treated as a 20-bit signed binary integer.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of LE, LD, LEY, and LDY)
- Data with DXC 1, AFP register
- Operation (LEY and LDY, if the long-displacement facility is not installed)
- Specification (LXR only)

STORE

Mnemonic  

\[
\begin{array}{cccccc}
\text{Op Code} & R_1 & X_2 & B_2 & D_2 \\
0 & 8 & 12 & 16 & 20 & 31
\end{array}
\]

The first operand is placed unchanged in storage at the second-operand location.

The displacement for STE and STD is treated as a 12-bit unsigned binary integer. The displacement for STEY and STDY is treated as a 20-bit signed binary integer.
**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Data with DXC 1, AFP register
- Operation (STEY and STDY, if the long-displacement facility is not installed)

---

**Summary of All Floating-Point Instructions**

Figures 9-9 through 9-13 on following pages show all floating-point instructions arranged in various categories of operand format and type of operation (principally, register-and-register and register-and-storage operations). Figure 9-9 shows the floating-point-support instructions. Figure 9-10 on page 9-13 shows the BFP and HFP instructions with all operands of the same length. Figure 9-11 on page 9-13 shows the BFP and HFP instructions in which the result is longer than the source operand. Figure 9-12 on page 9-14 shows the BFP and HFP instructions in which the result is shorter than the source operand. Figure 9-13 on page 9-14 shows the other BFP instructions, including those instructions which operate on the FPC register. The instructions CONVERT FROM FIXED and CONVERT TO FIXED convert between fixed-point and floating-point formats. In the figures, entries for 32-bit fixed-point operands are combined in the same column with entries for 32-bit short operands, and entries for 64-bit fixed-point operands are combined in the same column with entries for 64-bit long operands.

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Short (32)</th>
<th>Long (64)</th>
<th>Ext. (128)</th>
<th>32 to 64</th>
<th>64 to 32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R-R</td>
<td>R-S</td>
<td>R-R</td>
<td>R-R</td>
<td>R-R</td>
</tr>
<tr>
<td>CONVERT BFP TO HFP</td>
<td>THDR</td>
<td>R-R</td>
<td>THDR</td>
<td>R-R</td>
<td>R-R</td>
</tr>
<tr>
<td>CONVERT HFP TO BFP</td>
<td>TBDR</td>
<td>R-S</td>
<td>TBEDR</td>
<td>R-S</td>
<td>R-S</td>
</tr>
<tr>
<td>LOAD</td>
<td>LER</td>
<td>LE</td>
<td>LDR</td>
<td>LD</td>
<td>LXR</td>
</tr>
<tr>
<td>LOAD ZERO</td>
<td>LZER</td>
<td>LZDR</td>
<td>LZXR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>STE</td>
<td>Std</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**
- R-R Register-and-register operation.
- R-S Register-and-storage operation.

*Figure 9-9. Floating-Point-Support Instructions*
### Figure 9-10. BFP and HFP Instructions with All Operands of Same Length

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>HFP Instructions</th>
<th>BFP Instructions</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HFP Instructions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BFP Instructions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instruction Name</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HFP Instructions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BFP Instructions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD NORMALIZED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD UNNORMALIZED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPARE AND SIGNAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED&lt;sup&gt;1&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED&lt;sup&gt;1&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVIDE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVIDE TO INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD AND TEST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD COMPLEMENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD FP INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD NEGATIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTIPLY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTIPLY AND ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQUARE ROOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBTRACT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBTRACT UNNORMALIZED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST DATA CLASS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

1. This instruction also has mixed-length operands.

R-R Register-and-register operation.

R-S Register-and-storage operation.

### Figure 9-11. BFP and HFP Instructions with Result Longer than Source

| Instruction Name       | HFP Instructions | BFP Instructions |          |          |          |          |          |          |          |          |          |          |          |          |          |
|------------------------|------------------|------------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| **Instruction Name**   |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| **HFP Instructions**   |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| **BFP Instructions**   |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| **Instruction Name**   |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| **HFP Instructions**   |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| **BFP Instructions**   |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| CONVERT FROM FIXED     |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| CONVERT TO FIXED       |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| LOAD LENGTHENED        |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |
| MULTIPLY               |                  |                  |          |          |          |          |          |          |          |          |          |          |          |          |

**Explanation:**

R-R Register-and-register operation.

R-S Register-and-storage operation.
### Figure 9-12. BFP and HFP Instructions with Result Shorter than Source

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>HFP Instructions</th>
<th>BFP Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64 to 32</td>
<td>128 to 64</td>
</tr>
<tr>
<td></td>
<td>R-R</td>
<td>R-R</td>
</tr>
<tr>
<td>CONVERT FROM FIXED</td>
<td>CEGR</td>
<td>CEGBR</td>
</tr>
<tr>
<td>CONVERT TO FIXED</td>
<td>CFDR</td>
<td>CGXR</td>
</tr>
<tr>
<td>LOAD ROUNDED</td>
<td>LEDR</td>
<td>LDXR</td>
</tr>
</tbody>
</table>

**Explanation:**
- **R-R** Register-and-register operation.

### Figure 9-13. Other BFP Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTRACT FPC</td>
<td>EFPC</td>
</tr>
<tr>
<td>LOAD FPC</td>
<td>LFPC</td>
</tr>
<tr>
<td>SET FPC</td>
<td>SFPC</td>
</tr>
<tr>
<td>SET ROUNDING MODE</td>
<td>SRNM</td>
</tr>
<tr>
<td>STORE FPC</td>
<td>STFPC</td>
</tr>
</tbody>
</table>
## Chapter 10. Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH AND SET AUTHORITY</td>
<td>10-6</td>
</tr>
<tr>
<td>BRANCH AND STACK</td>
<td>10-10</td>
</tr>
<tr>
<td>BRANCH IN SUBSPACE GROUP</td>
<td>10-13</td>
</tr>
<tr>
<td>COMPARE AND SWAP AND PURGE</td>
<td>10-18</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>10-19</td>
</tr>
<tr>
<td>EXTRACT AND SET EXTENDED AUTHORITY</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT PRIMARY ASN</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT PRIMARY ASN AND INSTANCE</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT SECONDARY ASN</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT SECONDARY ASN AND INSTANCE</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT STACKED REGISTERS</td>
<td>10-22</td>
</tr>
<tr>
<td>EXTRACT STACKED STATE</td>
<td>10-24</td>
</tr>
<tr>
<td>INSERT ADDRESS SPACE CONTROL</td>
<td>10-27</td>
</tr>
<tr>
<td>INSERT PSW KEY</td>
<td>10-28</td>
</tr>
<tr>
<td>INSERT STORAGE KEY EXTENDED</td>
<td>10-28</td>
</tr>
<tr>
<td>INSERT VIRTUAL STORAGE KEY</td>
<td>10-29</td>
</tr>
<tr>
<td>INVALIDATE DAT TABLE ENTRY</td>
<td>10-30</td>
</tr>
<tr>
<td>INVALIDATE PAGE TABLE ENTRY</td>
<td>10-34</td>
</tr>
<tr>
<td>LOAD ADDRESS SPACE PARAMETERS</td>
<td>10-36</td>
</tr>
<tr>
<td>LOAD CONTROL</td>
<td>10-45</td>
</tr>
<tr>
<td>LOAD PSW</td>
<td>10-45</td>
</tr>
<tr>
<td>LOAD PSW EXTENDED</td>
<td>10-46</td>
</tr>
<tr>
<td>LOAD REAL ADDRESS</td>
<td>10-47</td>
</tr>
<tr>
<td>LOAD USING REAL ADDRESS</td>
<td>10-52</td>
</tr>
<tr>
<td>MODIFY STACKED STATE</td>
<td>10-52</td>
</tr>
<tr>
<td>MOVE PAGE</td>
<td>10-54</td>
</tr>
<tr>
<td>MOVE TO PRIMARY</td>
<td>10-56</td>
</tr>
<tr>
<td>MOVE TO SECONDARY</td>
<td>10-56</td>
</tr>
<tr>
<td>MOVE WITH DESTINATION KEY</td>
<td>10-58</td>
</tr>
<tr>
<td>MOVE WITH KEY</td>
<td>10-59</td>
</tr>
<tr>
<td>MOVE WITH SOURCE KEY</td>
<td>10-60</td>
</tr>
<tr>
<td>PAGE IN</td>
<td>10-61</td>
</tr>
<tr>
<td>PAGE OUT</td>
<td>10-62</td>
</tr>
<tr>
<td>PROGRAM CALL</td>
<td>10-63</td>
</tr>
<tr>
<td>PROGRAM RETURN</td>
<td>10-76</td>
</tr>
<tr>
<td>PROGRAM TRANSFER</td>
<td>10-81</td>
</tr>
<tr>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>10-81</td>
</tr>
<tr>
<td>PURGE ALB</td>
<td>10-91</td>
</tr>
<tr>
<td>PURGE TLB</td>
<td>10-91</td>
</tr>
<tr>
<td>RESET REFERENCE BIT EXTENDED</td>
<td>10-91</td>
</tr>
<tr>
<td>RESUME PROGRAM</td>
<td>10-92</td>
</tr>
<tr>
<td>SET ADDRESS SPACE CONTROL</td>
<td>10-95</td>
</tr>
<tr>
<td>SET ADDRESS SPACE CONTROL FAST</td>
<td>10-95</td>
</tr>
<tr>
<td>SET CLOCK</td>
<td>10-96</td>
</tr>
<tr>
<td>SET CLOCK COMPARATOR</td>
<td>10-97</td>
</tr>
<tr>
<td>SET CLOCK PROGRAMMABLE FIELD</td>
<td>10-97</td>
</tr>
<tr>
<td>SET CPU TIMER</td>
<td>10-98</td>
</tr>
<tr>
<td>SET PREFIX</td>
<td>10-98</td>
</tr>
<tr>
<td>SET PSW KEY FROM ADDRESS</td>
<td>10-99</td>
</tr>
<tr>
<td>SET SECONDARY ASN</td>
<td>10-99</td>
</tr>
<tr>
<td>SET SECONDARY ASN WITH INSTANCE</td>
<td>10-99</td>
</tr>
<tr>
<td>SET STORAGE KEY EXTENDED</td>
<td>10-105</td>
</tr>
<tr>
<td>SET SYSTEM MASK</td>
<td>10-105</td>
</tr>
<tr>
<td>SIGNAL PROCESSOR</td>
<td>10-106</td>
</tr>
<tr>
<td>STORE CLOCK COMPARATOR</td>
<td>10-107</td>
</tr>
<tr>
<td>STORE CONTROL</td>
<td>10-107</td>
</tr>
<tr>
<td>STORE CPU ADDRESS</td>
<td>10-108</td>
</tr>
<tr>
<td>STORE CPU ID</td>
<td>10-108</td>
</tr>
<tr>
<td>STORE CPU TIMER</td>
<td>10-109</td>
</tr>
<tr>
<td>STORE FACILITY LIST</td>
<td>10-110</td>
</tr>
<tr>
<td>STORE PREFIX</td>
<td>10-110</td>
</tr>
<tr>
<td>STORE REAL ADDRESS</td>
<td>10-111</td>
</tr>
<tr>
<td>STORE SYSTEM INFORMATION</td>
<td>10-112</td>
</tr>
<tr>
<td>STORE THEN AND SYSTEM MASK</td>
<td>10-124</td>
</tr>
<tr>
<td>STORE THEN OR SYSTEM MASK</td>
<td>10-124</td>
</tr>
<tr>
<td>STORE USING REAL ADDRESS</td>
<td>10-124</td>
</tr>
<tr>
<td>TEST ACCESS</td>
<td>10-125</td>
</tr>
<tr>
<td>TEST BLOCK</td>
<td>10-127</td>
</tr>
<tr>
<td>TEST PROTECTION</td>
<td>10-129</td>
</tr>
<tr>
<td>TRACE</td>
<td>10-132</td>
</tr>
<tr>
<td>TRAP</td>
<td>10-133</td>
</tr>
</tbody>
</table>

This chapter includes all privileged and semiprivileged instructions described in this publication, except the input/output instructions, which are described in Chapter 14, "I/O Instructions."

Privileged instructions may be executed only when the CPU is in the supervisor state. An attempt to execute a privileged instruction in the problem state generates a privileged-operation exception.

The semiprivileged instructions are those instructions that can be executed in the problem state when certain authority requirements are met. An attempt to execute a semiprivileged instruction in the problem state when the authority requirements are not met generates a privileged-operation exception or some other program-interruption condition depending on the particular
requirement which is violated. Those requirements which cause a privileged-operation exception to be generated in the problem state are not enforced when execution is attempted in the supervisor state.

The control instructions and their mnemonics, formats, and operation codes are listed in Figure 10-1 on page 10-3. The figure also indicates which instructions are new in z/Architecture as compared to ESA/390, when the condition code is set, the instruction fields that designate access registers, and the exceptional conditions in operand designations, data, or results that cause a program interruption.

The instructions that are new in z/Architecture are indicated in Figure 10-1 by “N.”

When the operands of an instruction are 32-bit operands, the mnemonic for the instruction does not include a letter indicating the operand length. If there is an instruction with the same name but with 64-bit operands, its mnemonic includes the letter “G.” In Figure 10-1, when there is an instruction with 32-bit operands and another instruction with the same name but with “G” added in its mnemonic, the first instruction has “(32)” after its name, and the other instruction has “(64)” after its name.

For those control instructions which have special rules regarding the handling of exceptional situations, a section called “Special Conditions” is included. This section indicates the type of ending (suppression, nullification, or completion) only for those exceptions for which the ending may vary.

Note: In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designation for the assembler language are shown with each instruction. For LOAD PSW, for example, LPSW is the mnemonic and $D_2(B_2)$ the operand designation.

Programming Notes:

1. The following additional control instructions are available when the DAT-enhancement facility is installed:
   - COMPARE AND SWAP AND PURGE (CSPG)
   - INVALIDATE DAT TABLE ENTRY
   CSPG operates on a doubleword in storage, in contrast to the previously existing instruction COMPARE AND SWAP AND PURGE (CSP), which operates on a word in storage.

2. The long-displacement facility uses new instruction formats, named RSY, RXY, and SIY, to provide 20-bit signed displacements. In connection with the long-displacement facility, all previously existing control instructions of the RSE or RXE format are changed to be of format RSY or RXY, respectively, where the new formats differ from the old by using a previously unused byte, now named DH, in the instructions. When the long-displacement facility is installed, the displacement for an instruction operand address is formed by appending DH on the left of the previous displacement field, now named DL, of the instruction. When the long-displacement facility is not installed, eight zero bits are appended on the left of DL, and DH is ignored.

   The following additional control instruction is available when the long-displacement facility is installed:
   - LOAD REAL ADDRESS (LRAY)

   The following additional control instructions are available when the ASN-and-LX-reuse facility is installed:
   - EXTRACT PRIMARY ASN AND INSTANCE
   - EXTRACT SECONDARY ASN AND INSTANCE
   - PROGRAM TRANSFER WITH INSTANCE
   - SET SECONDARY ASN WITH INSTANCE
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH AND SET AUTHORITY</td>
<td>BSA</td>
<td>RRE</td>
<td>Q A 1</td>
<td>SO T</td>
</tr>
<tr>
<td>BRANCH AND STACK</td>
<td>BAKR</td>
<td>RRE</td>
<td>A 1</td>
<td>Z 2</td>
</tr>
<tr>
<td>BRANCH IN SUBSPACE GROUP</td>
<td>BSG</td>
<td>RRE</td>
<td>A 1</td>
<td>SO T</td>
</tr>
<tr>
<td>COMPARE AND SWAP AND PURGE</td>
<td>CSPG</td>
<td>RRE C DE</td>
<td>P A 1 SP</td>
<td>$</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td>DM</td>
<td></td>
<td>P DM</td>
<td>MD</td>
</tr>
<tr>
<td>EXTRACT AND SET EXTENDED AUTHORITY</td>
<td>ESEA</td>
<td>RRE N</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>EXTRACT PRIMARY ASN</td>
<td>EPAR</td>
<td>RRE</td>
<td>Q SO</td>
<td>B226</td>
</tr>
<tr>
<td>EXTRACT PRIMARY ASN AND INSTANCE</td>
<td>EPAIR</td>
<td>RRE RA</td>
<td>Q SO</td>
<td>B99A</td>
</tr>
<tr>
<td>EXTRACT SECONDARY ASN</td>
<td>ESAR</td>
<td>RRE Q SO</td>
<td></td>
<td>B227</td>
</tr>
<tr>
<td>EXTRACT SECONDARY ASN AND INSTANCE</td>
<td>ESAIR</td>
<td>RRE RA</td>
<td>Q SO</td>
<td>U1</td>
</tr>
<tr>
<td>EXTRACT STACKED REGISTERS (32)</td>
<td>EREG</td>
<td>RRE</td>
<td>A 1</td>
<td>SE</td>
</tr>
<tr>
<td>EXTRACT STACKED REGISTERS (64)</td>
<td>EREGG</td>
<td>RRE</td>
<td>A 1</td>
<td>SE</td>
</tr>
<tr>
<td>EXTRACT STACKED STATE</td>
<td>ESTA</td>
<td>RRE C SP</td>
<td></td>
<td>B244</td>
</tr>
<tr>
<td>INSERT ADDRESS SPACE CONTROL</td>
<td>IAC</td>
<td>RRE</td>
<td>Q</td>
<td>ST</td>
</tr>
<tr>
<td>INSERT PSW KEY</td>
<td>IPK</td>
<td>S</td>
<td>Q</td>
<td>G2</td>
</tr>
<tr>
<td>INSERT STORAGE KEY EXTENDED</td>
<td>ISKE</td>
<td>RRE P A 1</td>
<td></td>
<td>B229</td>
</tr>
<tr>
<td>INSERT VIRTUAL STORAGE KEY</td>
<td>IVSK</td>
<td>RRE Q A 1</td>
<td>SO</td>
<td>R2</td>
</tr>
<tr>
<td>INVALIDATE DAT TABLE ENTRY</td>
<td>IOTE</td>
<td>RRE DE</td>
<td>P A 1</td>
<td>$</td>
</tr>
<tr>
<td>INVALIDATE PAGE TABLE ENTRY</td>
<td>IPTE</td>
<td>RRE P A 1</td>
<td>$</td>
<td>B221</td>
</tr>
<tr>
<td>LOAD ADDRESS SPACE PARAMETERS</td>
<td>LASP</td>
<td>SSE</td>
<td>P A 1 SP</td>
<td>SO</td>
</tr>
<tr>
<td>LOAD CONTROL (32)</td>
<td>LCTL</td>
<td>RS</td>
<td>P A SP</td>
<td></td>
</tr>
<tr>
<td>LOAD CONTROL (64)</td>
<td>LCTLG</td>
<td>RSY N</td>
<td>P A SP</td>
<td></td>
</tr>
<tr>
<td>LOAD PSW</td>
<td>LPSW</td>
<td>S L P A SP ¢</td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td>LOAD PSW EXTENDED</td>
<td>LPSWE</td>
<td>S L N P A SP ¢</td>
<td></td>
<td>B222</td>
</tr>
<tr>
<td>LOAD REAL ADDRESS (32)</td>
<td>LRA</td>
<td>RX C P A 1</td>
<td>SO</td>
<td>B1</td>
</tr>
<tr>
<td>LOAD REAL ADDRESS (64)</td>
<td>LRAG</td>
<td>RX N P A 1</td>
<td>SO</td>
<td>B243</td>
</tr>
<tr>
<td>LOAD USING REAL ADDRESS (32)</td>
<td>LURA</td>
<td>RRE P A 1 SP</td>
<td></td>
<td>B24B</td>
</tr>
<tr>
<td>LOAD USING REAL ADDRESS (64)</td>
<td>LURAG</td>
<td>RRE N P A 1 SP</td>
<td></td>
<td>B905</td>
</tr>
<tr>
<td>MODIFY STACKED STATE</td>
<td>MSTA</td>
<td>RRE</td>
<td>A 1 SP</td>
<td>SE</td>
</tr>
<tr>
<td>MOVE PAGE</td>
<td>MVPG</td>
<td>RRE C Q A SP</td>
<td>G0</td>
<td>R1</td>
</tr>
<tr>
<td>MOVE TO PRIMARY</td>
<td>MVC</td>
<td>SS C Q A SO ¢</td>
<td>ST</td>
<td>DA</td>
</tr>
<tr>
<td>MOVE TO SECONDARY</td>
<td>MVCS</td>
<td>SS C Q A SO ¢</td>
<td>ST</td>
<td>DB</td>
</tr>
<tr>
<td>MOVE WITH DESTINATION KEY</td>
<td>MVDK</td>
<td>SSE Q A</td>
<td>GM</td>
<td>B1</td>
</tr>
<tr>
<td>MOVE WITH KEY</td>
<td>MVCK</td>
<td>SS C Q A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MOVE WITH SOURCE KEY</td>
<td>MVCSK</td>
<td>SSE Q A</td>
<td>GM</td>
<td>B1</td>
</tr>
<tr>
<td>PAGE IN</td>
<td>PGIN</td>
<td>RRE C ES P A 1</td>
<td>ST</td>
<td>B22E</td>
</tr>
<tr>
<td>PAGE OUT</td>
<td>PGOUT</td>
<td>RRE C ES P A 1</td>
<td>ST</td>
<td>B22F</td>
</tr>
<tr>
<td>PROGRAM CALL</td>
<td>PC</td>
<td>S Q A Z T ¢</td>
<td>GM</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 10-1 (Part 1 of 4). Summary of Control Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM RETURN</td>
<td>PR</td>
<td>E L Q A SP</td>
<td>Z^2 T \epsilon^2</td>
</tr>
<tr>
<td>PROGRAM TRANSFER</td>
<td>PT</td>
<td>RRE Q A SP \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>PTI</td>
<td>RRE RA Q A SP \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>PURGE ALB</td>
<td>PALB</td>
<td>P $</td>
<td>B</td>
</tr>
<tr>
<td>PURGE TLB</td>
<td>PTLB</td>
<td>S $</td>
<td>B</td>
</tr>
<tr>
<td>RESET REFERENCE BIT EXTENDED</td>
<td>RRBE</td>
<td>RRE C P A SP</td>
<td>B</td>
</tr>
<tr>
<td>RESUME PROGRAM</td>
<td>RP</td>
<td>S L Q A SP WE T</td>
<td>B</td>
</tr>
<tr>
<td>SET ADDRESS SPACE CONTROL</td>
<td>SAC</td>
<td>S Q SP SW \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>SET ADDRESS SPACE CONTROL FAST</td>
<td>SACF</td>
<td>S Q SP SW \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>SET CLOCK</td>
<td>SCK</td>
<td>S C P A SP</td>
<td>B</td>
</tr>
<tr>
<td>SET CLOCK COMPARATOR</td>
<td>SCKC</td>
<td>S P A SP</td>
<td>B</td>
</tr>
<tr>
<td>SET CLOCK PROGRAMMABLE FIELD</td>
<td>SCKPF</td>
<td>E P SP G0</td>
<td>B</td>
</tr>
<tr>
<td>SET CPU TIMER</td>
<td>SPT</td>
<td>S P A SP</td>
<td>B</td>
</tr>
<tr>
<td>SET PREFIX</td>
<td>SPX</td>
<td>S P A SP $</td>
<td>B</td>
</tr>
<tr>
<td>SET PSW KEY FROM ADDRESS</td>
<td>SPKA</td>
<td>S Q</td>
<td>B</td>
</tr>
<tr>
<td>SET SECONDARY ASN</td>
<td>SSAR</td>
<td>RRE A SP Z^3 T \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>SET SECONDARY ASN WITH INSTANCE</td>
<td>SSAIR</td>
<td>RRE RA A SP Z^7 T \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>SET STORAGE KEY EXTENDED</td>
<td>SSK</td>
<td>RRE P A SP \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>SET SYSTEM MASK</td>
<td>SSM</td>
<td>S P A SP SO</td>
<td>B</td>
</tr>
<tr>
<td>SIGNAL PROCESSOR</td>
<td>SIGP</td>
<td>S RS C P</td>
<td>B</td>
</tr>
<tr>
<td>STORE CLOCK COMPARATOR</td>
<td>STCKC</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE CONTROL (32)</td>
<td>STCTL</td>
<td>S RS P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE CONTROL (64)</td>
<td>STCTG</td>
<td>S RSN P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE CPU ADDRESS</td>
<td>STAP</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE CPU ID</td>
<td>STIDP</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE CPU TIMER</td>
<td>STPT</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE FACILITY LIST</td>
<td>STFL</td>
<td>S P A SP N3</td>
<td>ST</td>
</tr>
<tr>
<td>STORE PREFIX</td>
<td>STPX</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE REAL ADDRESS</td>
<td>STRAG</td>
<td>SSE N P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE SYSTEM INFORMATION</td>
<td>STSI</td>
<td>S C P A SP GM</td>
<td>ST</td>
</tr>
<tr>
<td>STORE THEN AND SYSTEM MASK</td>
<td>STNSM</td>
<td>S P A</td>
<td>ST</td>
</tr>
<tr>
<td>STORE THEN OR SYSTEM MASK</td>
<td>STDMS</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE USING REAL ADDRESS (32)</td>
<td>STURA</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>STORE USING REAL ADDRESS (64)</td>
<td>STURG</td>
<td>S P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>TEST ACCESS</td>
<td>STRTG</td>
<td>S RRE C A SP</td>
<td>ST</td>
</tr>
<tr>
<td>TEST BLOCK</td>
<td>TB</td>
<td>RRE C P A SP II $ G0</td>
<td>B</td>
</tr>
<tr>
<td>TEST PROTECTION</td>
<td>TPROT</td>
<td>S SSE C P A SP</td>
<td>B</td>
</tr>
<tr>
<td>TRACE (32)</td>
<td>TRACE</td>
<td>RRE RA SP T \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>TRACE (64)</td>
<td>TRACP</td>
<td>RRE RA P A SP T \epsilon</td>
<td>B</td>
</tr>
<tr>
<td>TRAP</td>
<td>TRAP2</td>
<td>E A SO T ST</td>
<td>B</td>
</tr>
<tr>
<td>TRAP</td>
<td>TRAP4</td>
<td>S A SO T ST</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure 10-1 (Part 2 of 4). Summary of Control Instructions
**Explanation:**

- $\xi$ Causes serialization and checkpoint synchronization.
- $\xi^2$ Causes serialization and checkpoint synchronization when the state entry to be unstacked is a program-call state entry.
- $\xi$ Causes serialization.
- A Access exceptions for logical addresses.
- $A^1$ Access exceptions; not all access exceptions may occur; see instruction description for details.
- B PER branch event.
- $B_1$ $B_1$ field designates an access register in the access-register mode.
- $B_2$ $B_2$ field designates an access register in the access-register mode.
- BP $BP$ field designates an access register when PSW bits 16 and 17 have the value 01 binary.
- C Condition code is set.
- DE DAT-enhancement facility.
- DM Depending on the model, DIAGNOSE may generate various program exceptions and may change the condition code.
- ES Expanded-storage facility.
- FC Designation of access registers depends on the function code of the instruction.
- G0 Instruction execution includes the implied use of general register 0.
- G2 Instruction execution includes the implied use of general register 2.
- GM Instruction execution includes the implied use of multiple general registers:
  - General registers 0 and 1 for MOVE WITH DESTINATION KEY, MOVE WITH SOURCE KEY, and STORE SYSTEM INFORMATION.
  - General registers 3, 4, and 14, and, when the ASN-and-LX-reuse facility is installed, general register 15 for PROGRAM CALL.
- II Interruptible instruction.
- L New condition code is loaded.
- LD Long-displacement facility.
- MD Designation of access registers in the access-register mode is model-dependent.
- N Instruction is new in z/Architecture as compared to ESA/390.
- N3 Instruction is new in z/Architecture and has been added to ESA/390.
- P Privileged-operation exception.
- Q Privileged-operation exception for semiprivileged instructions.
- $R_1$ $R_1$ field designates an access register in the access-register mode.
- $R_2$ $R_2$ field designates an access register in the access-register mode.
- RA Reusable-ASN-and-LX facility.
- RRE RRE instruction format.
- RS RS instruction format.
- RSY RSY instruction format.
- RX RX instruction format.
- RXY RXY instruction format.
- S S instruction format.
- SE Special-operation, stack-empty, stack-specification, and stack-type exceptions.
- SF Special-operation, stack-full, and stack-specification exceptions.
- SI SI instruction format.
- SO Special-operation exception.
- SP Specification exception.
- SS SS instruction format.
- SSE SSE instruction format.
- ST PER storage-alteration event.
- SU PER store-using-real-address event.
- SW Special-operation exception and space-switch event.
- T Trace exceptions (which include trace table, addressing, and low-address protection).

**Figure 10-1 (Part 3 of 4). Summary of Control Instructions**
Explanation (Continued):

U1 R1 field designates an access register unconditionally.
U2 R2 field designates an access register unconditionally.
WE Space-switch event.
Z1 Additional exceptions and events for PROGRAM CALL (which include ASX-translation, EX-translation, LFX-translation, LSTE-sequence, LSX-translation, LX-translation, PC-translation-specification, special-operation, stack-full, stack-specification and subspace-replacement exceptions and space-switch event).
Z2 Additional exceptions and events for PROGRAM TRANSFER (which include AFX-translation, ASX-translation, primary-authority, special-operation, and subspace-replacement exceptions and space-switch event).
Z3 Additional exceptions for SET SECONDARY ASN (which include AFX translation, ASX translation, secondary authority, special operation and subspace replacement).
Z4 Additional exceptions and events for PROGRAM RETURN (which include AFX-translation, ASTE-instance, ASX-translation, secondary-authority, special-operation, stack-empty, stack-operation, stack-specification, stack-type, and subspace-replacement exceptions and space-switch event).
Z5 Additional exceptions for BRANCH AND STACK (which include special operation, stack full, and stack specification).
Z6 Additional exceptions and events for PROGRAM TRANSFER WITH INSTANCE (which include AFX-translation, ASTE-instance, ASX-translation, primary-authority, special-operation, and subspace-replacement exceptions and space-switch event).
Z7 Additional exceptions for SET SECONDARY ASN WITH INSTANCE (which include AFX translation, ASTE instance, ASX translation, secondary authority, special operation, and subspace replacement).

Figure 10-1 (Part 4 of 4). Summary of Control Instructions

**BRANCH AND SET AUTHORITY**

BSA R1,R2 [RRE]

<table>
<thead>
<tr>
<th>'B25A'</th>
<th>//////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the dispatchable unit is in the base-authority state and the 24-bit or 31-bit addressing mode: bits 32 and 97-127 of the current PSW, the basic-addressing-mode bit and bits 33-63 of the updated instruction address, are saved in the dispatchable-unit control table (DUCT); the PSW-key mask (PKM), PSW key, and problem-state bit also are saved in the DUCT; the PKM and PSW key are replaced using the contents of general register R1; the problem-state bit is set to one; bits 32 and 97-127 of the PSW are replaced using the contents of general register R2; and the dispatchable unit is placed in the reduced-authority state. In the 64-bit addressing mode, the action is the same except that bits 64-127 of the current PSW are saved in general register R1 if R1 is not zero, those bits in the PSW are replaced from the DUCT, and bit 32 of the PSW is neither saved nor replaced.

Words 5, 8, and 9 of the DUCT are used by this instruction. The contents of those words are as follows:

<table>
<thead>
<tr>
<th>PSW-Key Mask</th>
<th>PSW Key</th>
<th>R</th>
<th>A</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 16 24 28 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the 24-Bit or 31-Bit Addressing Mode

<table>
<thead>
<tr>
<th>All Zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 33-63 of Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 31</td>
</tr>
</tbody>
</table>

in general register R1 if R1 is not zero; bits 32 and 97-127 of the PSW and the PKM, PSW key, and problem-state bit are replaced by values saved in the DUCT; and the dispatchable unit is placed in the base-authority state. In the 64-bit addressing mode, the action is the same except that bits 64-127 of the current PSW are saved in general register R1 if R1 is not zero, those bits in the PSW are replaced from the DUCT, and bit 32 of the PSW is neither saved nor replaced.
In the 64-Bit Addressing Mode

<table>
<thead>
<tr>
<th></th>
<th>Bits 0-31 of Return Address</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The fields in words 5, 8, and 9 of the DUCT are allocated as follows:

**PSW-Key Mask:** Bit positions 0-15 of word 5 contain the PSW-key mask (PKM), bits 32-47 of control register 3, saved by BRANCH AND SET AUTHORITY executed in the base-authority state. The PKM is restored to control register 3 by BRANCH AND SET AUTHORITY executed in the reduced-authority state.

**PSW Key:** Bit positions 24-27 of word 5 contain the PSW key, bits 8-11 of the PSW, saved by BRANCH AND SET AUTHORITY executed in the base-authority state. The PSW key is restored to the PSW by BRANCH AND SET AUTHORITY executed in the reduced-authority state.

**Reduced Authority (RA):** Bit 28 of word 5 indicates, when zero, that the dispatchable unit associated with the DUCT is in the base-authority state or, when one, that the dispatchable unit is in the reduced-authority state. Bit 28 is set to one by BRANCH AND SET AUTHORITY executed in the base-authority state, and it is set to zero by BRANCH AND SET AUTHORITY executed in the reduced-authority state.

**Problem State (P):** Bit position 31 of word 5 contains the problem-state bit, bit 15 of the PSW, saved by BRANCH AND SET AUTHORITY executed in the base-authority state. The problem-state bit is restored to the PSW by BRANCH AND SET AUTHORITY executed in the reduced-authority state.

**Basic Addressing Mode (BA):** In the 24-bit or 31-bit addressing mode, bit position 0 of word 9 contains the basic-addressing-mode bit, bit 32 of the PSW, saved by BRANCH AND SET AUTHORITY executed in the base-authority state. The basic-addressing-mode bit is restored to the PSW from the DUCT by BRANCH AND SET AUTHORITY executed in the reduced-authority state.

**Return Address:** In the 24-bit or 31-bit addressing mode, bit positions 1-31 of word 9 contain bits 33-63 of the updated instruction address, bits 97-127 of the PSW, saved by BRANCH AND SET AUTHORITY executed in the base-authority state. Bits 1-31 of word 9 of the DUCT are restored to bit positions 97-127 of the PSW by BRANCH AND SET AUTHORITY executed in the reduced-authority state. The contents of words 8 and 9 are restored to bit positions 64-127 of the PSW by BRANCH AND SET AUTHORITY executed in the reduced-authority state.

In the 24-bit or 31-bit addressing mode, all zeros are stored in word 8 when saving occurs in the base-authority state. In any addressing mode, all zeros are stored in bit positions 16-23, 29, and 30 of word 5 when saving occurs in the base-authority state.

All other fields in words 5, 8, and 9 remain unchanged when bit 28 of word 5 is set to zero in the reduced-authority state.

The fetch, store, and update references to the DUCT are single-access references and appear to be word concurrent as observed by other CPUs. The words of the DUCT are accessed in no particular order.

**Base-Authority Operation**

When BRANCH AND SET AUTHORITY is executed in the base-authority state, as indicated by the reduced-authority bit (RA) in the DUCT being zero, \( R_2 \) must be nonzero; otherwise, a special-operation exception is recognized. \( R_1 \) may be zero or nonzero.

The contents of bit positions 32-63 of general register \( R_1 \) and of general register \( R_2 \) when the execution of the instruction begins in the base-authority state are as follows:
In any addressing mode, the contents of bit positions 0-31 of general register R₁ are ignored. In the 24-bit or 31-bit addressing mode, the contents of bit positions 0-31 of general register R₂ are ignored.

In the 24-bit or 31-bit addressing mode, PSW bits 32 and 97-127 are saved in word 9 of the DUCT, and zeros are stored in word 8. In the 64-bit addressing mode, PSW bits 64-127 are saved in words 8 and 9 of the DUCT. In any addressing mode, the PKM, the PSW key, and the problem-state bit are saved in word 5 of the DUCT, the RA bit in word 5 is set to one, and bits 16-23, 29, and 30 of word 5 are set to zeros.

Bits 56-59 of general register R₁ are placed in bit positions 8-11 of the PSW as the new PSW key. In the problem state, the new PSW key must be authorized by the PKM; otherwise, if the new PSW key is not authorized, a privileged-operation exception is recognized.

After the new PSW key has been placed in the PSW, bits 32-47 of general register R₁ are ANDed with the PKM in control register 3, and the result replaces the PKM in control register 3.

The problem-state bit in the PSW is set to one.

In the 24-bit or 31-bit addressing mode, bit 32 of general register R₂ is placed in bit position 32 of the PSW as the new basic-addressing-mode bit. A branch address is generated from bits 33-63 of general register R₂ under the control of the new basic addressing mode, and the result is placed in bit positions 64-127 of the PSW as the new instruction address.

In the 64-bit addressing mode, a branch address is generated from bits 0-63 of general register R₂ and is placed in bit positions 64-127 of the PSW as the new instruction address. Bit 32 of the PSW remains unchanged.

Bits 48-55 and 60-63 of general register R₁ may be used for future extensions and should be zeros; otherwise, the program may not operate compatibly in the future.

Reduced-Authority Operation

When BRANCH AND SET AUTHORITY is executed in the reduced-authority state, as indicated by the reduced-authority (RA) bit in the DUCT being one, R₂ must be zero; otherwise, a special-operation exception is recognized. R₁ may be zero or nonzero. The initial contents of general registers R₁ and R₂ are ignored.

If R₁ is nonzero in the 24-bit or 31-bit addressing mode, bits 32 and 97-127 of the current PSW, the basic-addressing-mode bit and bits 33-63 of the updated instruction address, are placed in bit positions 32 and 33-63, respectively, of general register R₁, and bits 0-31 of the register remain unchanged. If R₁ is nonzero in the 64-bit addressing mode, bits 64-127 of the current PSW are placed in bit positions 0-63 of general register R₁. If R₁ is zero, general register 0 remains unchanged.

In the 24-bit or 31-bit addressing mode, bit 0 of word 9 of the DUCT is placed in PSW bit position 32, and bits 1-31 of word 9, with 33 leftmost zeros appended, are placed in PSW bit positions 64-127.

In the 64-bit addressing mode, the contents of words 8 and 9 of the DUCT are placed in PSW bit positions 64-127, and bit 32 of the PSW remains unchanged.
In any addressing mode, the PKM, the PSW key, and the problem-state bit are restored from the DUCT, and the RA bit is set to zero, as previously described. There is no test for whether the restored PSW key is authorized by the restored PKM.

**Special Conditions**

R$_2$ must be nonzero in the base-authority state and zero in the reduced-authority state. If either of these rules is violated, a special-operation exception is recognized, and the operation is suppressed.

In the problem state, the execution of the instruction in the base-authority state is subject to control by the PSW-key mask in control register 3. When the bit in the PSW-key mask corresponding to the PSW-key value to be set is one, the instruction is executed successfully. When the selected bit in the PSW-key mask is zero, a privileged-operation exception is recognized. In the supervisor state, any value for the PSW key is valid.

Key-controlled protection does not apply to any access made during the operation. Low-address protection does apply.

In the 24-bit or 31-bit addressing mode, the contents of word 9 of the DUCT are not checked for validity before they are loaded into the PSW. However, after loading, a specification exception is recognized, and a program interruption occurs, when the newly loaded PSW contains a zero in bit position 32 and the contents of bit positions 97-103 are not all zeros. In this case, the operation is completed, and the resulting instruction-length code is 0. The specification exception, which in this case is listed as a program exception in this instruction, is described in "Early Exception Recognition" on page 6-9. It may be considered as occurring early in the process of preparing to execute the following instruction.

The operation is suppressed on all addressing and protection exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-2 on page 10-10.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Addressing (dispatchable-unit control table)
- Privileged operation (selected PSW-key-mask bit is zero in the problem state, base-authority operation only)
- Protection (low-address; dispatchable-unit control table)
- Special operation
- Specification
- Trace

**Programming Notes:**

1. BRANCH AND SET AUTHORITY can improve performance by replacing to-current-primary forms of PROGRAM TRANSFER (PT-cp) and basic (nonstacking) PROGRAM CALL (PC-cp) instructions. PT-cp and PC-cp are often used (within a single address space) to reduce the authority of the PSW-key mask (PKM) or change from supervisor state to problem state during a calling linkage made by PT-cp and then to restore the PKM authority or supervisor state during a return linkage made by PC-cp. Also, the PSW-key-setting operations of BRANCH AND SET AUTHORITY can be substituted for SET PSW KEY FROM ADDRESS instructions, and, since BRANCH AND SET AUTHORITY combines branching with PSW-key setting, it can be used to change the PSW key when branching from or to a fetch-protected program.

2. Only one base-authority state and one reduced-authority state are available to a dispatchable unit. Nested use of BRANCH AND SET AUTHORITY, that is, use within different subroutine levels, is not possible. The requirement that R$_2$ must be nonzero in the base-authority state and zero in the reduced-authority state provides detection of an attempt to use BRANCH AND SET AUTHORITY in the base-authority state when the dispatchable unit is already in the reduced-authority state because of a previous use of the instruction in the base-authority state.

3. BRANCH AND SET AUTHORITY in the base-authority state does not save an indication in the DUCT of whether the current addressing mode is the extended (64-bit) addressing mode or a basic (24-bit or 31-bit) addressing mode. The instruction, in either the base-authority state or the reduced-authority state,
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


8.A Trace exceptions.

8.B Protection exception (low-address protection) for access to dispatchable-unit control table.

8.C.1 Addressing exception for access to dispatchable-unit control table.

8.C.2 Special-operation exception due to R/basesupertwo being zero in the base-authority state or R/basesupertwo being nonzero in the reduced-authority state.

8.C.3 Privileged-operation exception due to selected PSW-key-mask bit being zero (base-authority operation only).

9. Specification exception due to bit 32 of the newly loaded PSW zero when bits 97-103 are not all zeros (reduced-authority operation only).

Figure 10-2. Priority of Execution: BRANCH AND SET AUTHORITY

does not cause a switch between the extended addressing mode and a basic addressing mode. In the reduced-authority state, the contents of words 8 and 9 of the DUCT are interpreted based only on the current addressing mode. If saving occurs in the 31-bit addressing mode and then restoring occurs in the 64-bit addressing mode, bit 0 of word 9 of the DUCT will be used as an address bit instead of as the basic-addressing-mode bit. If saving occurs in the 64-bit addressing mode and then restoring occurs in the 24-bit addressing mode, an early specification exception may be recognized, after the instruction execution is completed, because bits 97-103 of the PSW may be nonzero when bit 32 is zero.

4. The instruction may be referred to as BSA-ba or BSA-ra depending on whether it is executed in the base-authority state or the reduced-authority state, respectively.

BRANCH AND STACK

\[
\begin{array}{cccc}
\text{BAKR} & R_1, R_2 & [RRE] \\
\hline
'B240' & \hfill & R_1 & R_2 \\
0 & 16 & 24 & 28 & 31 \\
\end{array}
\]

A linkage-stack branch state entry is formed, and the current PSW, except with an unpredictable PER mask and with the addressing-mode bits and instruction address replaced from the first operand, is placed in the state entry. Subsequently, the updated instruction address in the current PSW is replaced from the second operand. Indications of the current addressing-mode bits and the new instruction address are placed in the state entry, and the PSW-key mask, PASN, SASN, EAX, and contents of general registers 0-15 and access registers 0-15 also are placed in the state entry. When the ASN-and-LX-reuse facility is installed and the ASN-and-LX-reuse control in control register 0 is one, the PASTEIN and SASTEIN also are placed in the state entry. The action associated with an operand is not performed if the R field designating the operand is zero.
When the \( R_1 \) field is nonzero, the contents of general register \( R_1 \) specify an address referred to as the return address.

When \( R_1 \) is nonzero and bit 63 of general register \( R_1 \) is zero, the return address is generated from the contents of the register under the control of the basic addressing mode specified by bit 32 of the register: 24-bit mode if bit 32 is zero, or 31-bit mode if bit 32 is one. Bit 32 of the register and the return address are substituted for the basic-addressing-mode bit, bit 32, and the updated instruction address, respectively, in the current PSW when the contents of that PSW are placed in the state entry. The extended-addressing-mode bit, bit 31, is set to zero in the PSW that is placed in the state entry. The contents of the current PSW are not changed.

When \( R_1 \) is nonzero and bit 63 of general register \( R_1 \) is one, the return address is generated from the contents of the register under the control of the 64-bit addressing mode. Bits 0-62 of the return address, with a zero appended on the right, are substituted for the updated instruction address in the current PSW when the contents of that PSW are placed in the state entry. The extended-addressing-mode bit, bit 31, is set to one in the PSW that is placed in the state entry. The contents of the current PSW are not changed.

When the \( R_1 \) field is zero, the current PSW is placed in the state entry without any change except for an unpredictable PER mask.

Subsequently, when the \( R_2 \) field is nonzero, the instruction address in the current PSW is replaced by the branch address. The branch address is generated from the contents of general register \( R_2 \) under the control of the current addressing mode. When the \( R_2 \) field is zero, the operation is performed without branching.

The branch state entry is formed and information is placed in it as described in “Stacking Process” on page 5-83.

In the 24-bit or 31-bit addressing mode, bits 33-63 of the branch address (or of the updated instruction address if the operation is performed without branching) are placed in bit positions 33-63 of bytes 144-151 in the state entry, bit 32 of the current PSW is placed in bit position 32 of those bytes, and zeros are placed in bit positions 0-31 of the bytes.

In the 64-bit addressing mode, bits 0-62 of the branch address (or of the updated instruction address if the operation is performed without branching) are placed in bit positions 0-62 of bytes 144-151 in the state entry, and a one is placed in bit position 63 of those bytes.

The entry-type code in the state entry is 0001100 binary.

Key-controlled protection does not apply to accesses to the linkage stack, but low-address and page protection do apply.

Special Conditions

The CPU must be in the primary-space mode or access-register mode; otherwise, a special-operation exception is recognized.

A stack-full or stack-specification exception may be recognized during the stacking process.

The operation is suppressed on all addressing and protection exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-3 on page 10-12.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch or store, except for key-controlled protection, linkage-stack entry)
- Special operation
- Stack full
- Stack specification
- Trace
1.-6. Exceptions with the same priority as the priority of program-
interruption conditions for the general case.


7.B Special-operation exception due to DAT being off or the CPU
being in secondary-space mode or home-space mode.

8.A Trace exceptions (only if R2 is nonzero).

8.B.1 Access exceptions (fetch) for entry descriptor of the current
linkage-stack entry.

Note: Exceptions 8.B.2-8.B.7 can occur only if there is not
enough remaining free space in the current linkage-stack
section.

8.B.2 Stack-specification exception due to remaining-free-space
value in current linkage-stack entry not being a multiple of
8.

8.B.3 Access exceptions (fetch) for second word of the trailer
entry of the current section. The entry is presumed to be a
trailer entry; its entry-type field is not examined.

8.B.4 Stack-full exception due to forward-section validity bit in
the trailer entry being zero.

8.B.5 Access exceptions (fetch) for entry descriptor of the header
entry of the next section. This entry is presumed to be a
header entry; its entry-type field is not examined.

8.B.6 Stack-specification exception due to not enough remaining
free space in the next section.

8.B.7 Access exceptions (store) for second word of the header entry
of the next section. If there is no exception, the header is
now called the current entry.

8.B.8 Access exceptions (store) for entry descriptor of the current
entry and for the new state entry.

Figure 10-3. Priority of Execution: BRANCH AND STACK

Programming Notes:

1. Examples of the use of the BRANCH AND
STACK instruction are given in Appendix A,
"Number Representation and Instruction-Use
Examples."

2. In no case does BRANCH AND STACK
change the current addressing mode.

3. The effect when the R1 field is zero is that the
return address, which would otherwise be
specified by the R1 general register, is the
address of the next sequential instruction. In
this case, BRANCH AND STACK provides a
program-linkage function that is comparable to
the function of BRANCH AND SAVE.

4. BRANCH AND STACK with a nonzero R1
field is intended for use at or near the entry
point of a called program. The program may
be called by means of BRANCH AND LINK
(BALR) or BRANCH AND SAVE (BAS or
BASR) from a program being executed in the
24-bit or 31-bit addressing mode, by means of
BRANCH AND SAVE AND SET MODE from a
program being executed in any addressing
mode, or by means of a BRANCH AND SET
MODE instruction located in a “glue module”
and being executed in any addressing mode.
In all of these cases when the nonzero R1
field of the calling instruction is the same as
the R1 field of BRANCH AND STACK, and
even when the addressing mode was changed during the calling linkage, BRANCH AND STACK correctly saves the addressing mode and return address of the calling program so that the subsequent execution of PROGRAM RETURN will return correctly to the calling program.

BRANCH IN SUBSPACE GROUP

BSG  R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>'B258'</th>
<th>/ / / / /</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Provided that the current primary address space is in the subspace group, if any, associated with the current dispatchable unit, the access-list-entry token (ALET) in access register R₂ is translated by means of a special form of access-register translation (ART) to locate a destination ASN-second-table entry (DASTE). If the DASTE specifies the base space of the subspace group, the primary ASCE (PASCE) in control register 1 is replaced by the ASCE in the DASTE. If the DASTE specifies a subspace of the group, bits 0-55 and 58-63 of the PASCE are replaced by the same bits of the ASCE in the DASTE. In either case, the following actions also occur.

In the 24-bit or 31-bit addressing mode, bits 32 and 97-127 of the current PSW, the basic-addressing-mode bit and bits 33-63 of the updated instruction address, are saved in bit positions 32 and 33-63, respectively, of general register R₁, and bits 0-31 of the register remain unchanged. Subsequently, the basic-addressing-mode bit and bits 33-63 of the instruction address in the current PSW are replaced from bit positions 32-63 of general register R₂, and bits 0-31 of the register are ignored.

In the 64-bit addressing mode, bits 64-127 of the current PSW, the updated instruction address, are saved in bit positions 0-63 of general register R₁. Subsequently, the instruction address in the current PSW is replaced from bit positions 0-63 of general register R₂. Bit 32 of the PSW remains unchanged.

In any addressing mode, general register 0 remains unchanged if the R₁ field is zero.

The secondary ASCE (SASCE) in control register 7 is set equal to the new PASCE in control register 1. Also, the secondary ASN (SASN), bits 48-63 of control register 3, is set equal to the primary ASN (PASN), bits 48-63 of control register 4. If the ASN-and-LX-reuse facility is installed and is enabled by the ASN-and-LX-reuse control in control register 0, the secondary ASTEIN (SASTEIN), bits 0-31 of control register 3, is set equal to the primary ASTEIN (PASTEIN), bits 0-31 of control register 4.

The current primary address space is in the subspace group for the dispatchable unit if the current primary-ASTE origin (PASTEO), bits 33-57 of control register 5, designates the ASTE for the base space of the group. The PASTEO designates the base-space ASTE if the PASTEO is equal to the base-ASTE origin (BASTEO), bits 1-25 of word 0 of the dispatchable-unit control table (DUCT). For determining whether the PASTEO equals the BASTEO, either the PASTEO may be compared to the BASTEO or the entire contents of bit positions 32-63 of control register 5 may be compared to the entire contents of word 0 of the DUCT.

Ordinary ART is described in "Access-Register-Translation Process" on page 5-57. The special ART performed by this instruction is contrasted to ordinary ART as follows:

1. The special ART is performed regardless of whether the CPU is in the access-register mode.

2. If the ALET being translated is 00000000 hex, called ALET 0, the DASTE is the ASTE for the base space. Bit 0 of the DASTE is ignored.

3. If the ALET is 00000001 hex, called ALET 1, the DASTE is the ASTE for the last subspace entered by the dispatchable unit by means of BRANCH IN SUBSPACE GROUP. That ASTE is designated by the subspace-ASTE origin (SSASTEO), bits 1-25 of word 1 of the DUCT. A special-operation exception is recognized if a subspace has not previously been entered, as indicated by that the SSASTEO is all zeros. An ASTE-validity exception is recognized if bit 0 of the DASTE is one. An ASTE-sequence exception is recognized if the ASTE sequence number (ASTESN) in the DASTE does not equal the subspace ASTESN (SSASTESN) in word 3 of...
the DUCT. The DASTE located because of ALET 1 is considered to specify a subspace even if, due to an error, the DASTE is the ASTE for the base space. That is, there is no comparison of the SSASTEO to the BASTEO.

4. If the ALET is other than ALET 0 and ALET 1, an ASTE is located by obtaining its origin from an access-list entry (ALE) in a way similar to ordinary ART, and the DASTE is that located ASTE. In this case, as in ordinary ART:
   - An ALET-specification exception is recognized if bits 0-6 of the ALET are not zeros.
   - An ALEN-translation exception is recognized if the ALE is outside the effective access list or bit 0 of the ALE is one.
   - An ASTE-validity exception is recognized if bit 0 of the DASTE is one.
   - An ASTE-sequence exception is recognized if the ASTE sequence number (ASTESN) in the DASTE does not equal the ASTESN in the ALE.

The operation differs from ordinary ART in that the ALE sequence number (ALESN) in the ALE is not compared to the ALESN in the ALET, and the private bit in the ALE is treated as zero. Thus, ALE-sequence and extended-authority exceptions cannot occur.

The fetch-only bit in the ALE is ignored.

When the ALET is other than ALET 0 and ALET 1, the special ART may be performed by using the ART-lookaside buffer (ALB).

The DASTE located due to an ALET other than ALET 0 and ALET 1 may be the ASTE for the base space of the subspace group associated with the dispatchable unit. The DASTE is the base-space ASTE if the DASTE origin (DASTEO) obtained from an ALE by ART equals the BASTEO in the DUCT. For determining whether the DASTEO equals the BASTEO, either the DASTEO may be compared to the BASTEO, or the DASTEO with one leftmost and six rightmost zeros appended may be compared to the entire contents of word 0 of the DUCT. If the DASTE is not the base-space ASTE, the DASTE is treated as the ASTE for a subspace of the dispatchable unit's subspace group provided that (1) the subspace-group bit, bit 54, in the ASCE in the DASTE is one, and (2) the DASTE does not specify the base space of another subspace group. The DASTE specifies the base space of another subspace group if the base-space bit, bit 31 of word 0 of the DASTE, is one. A special-operation exception is recognized if either of those two provisions is not met.

If the DASTE specifies the base space of the subspace group, the PASCE in control register 1 is replaced by the ASCE in the DASTE. If the DASTE specifies a subspace, bits 0-55 and 58-63 of the PASCE are replaced by the same bits of the ASCE in the DASTE, and bits 56 and 57 of the PASCE, the storage-alteration-event bit and space-switch-event-control bit, remain unchanged.

If R₁ is nonzero in the 24-bit or 31-bit addressing mode, bits 32 and 97-127 of the current PSW, the basic-addressing-mode bit and bits 33-63 of the updated instruction address, are placed in bit positions 32 and 33-63, respectively, of general register R₁, and bits 0-31 of the register remain unchanged. If R₁ is nonzero in the 64-bit addressing mode, bits 64-127 of the current PSW, the updated instruction address, are placed in bit positions 0-63 of general register R₁. If R₁ is zero, general register 0 remains unchanged.

Whether R₂ is nonzero or zero, in the 24-bit or 31-bit addressing mode, bits 32-63 of general register R₂ specify the new basic addressing mode and designate the branch address. Bit 32 of the register specifies the new basic addressing mode and replaces bit 32 of the current PSW, and the branch address is generated from the contents of bit positions 33-63 of the register under the control of the new basic addressing mode.

When R₂ is nonzero or zero in the 64-bit addressing mode, the contents of general register R₂ designate the branch address. The branch address is generated from the contents of the register under the control of the 64-bit addressing mode. Bit 32 of the PSW remains unchanged.

Regardless of the addressing mode, the new value for the PSW is computed before general register R₁ is changed.

The secondary ASCE (SASCE) in control register 7 is set equal to the new PASCE in control register 1. The secondary ASN (SASN), bits 48-63 of control register 3, is set equal to the primary ASN (PASN), bits 48-63 of control register 4. If the ASN-and-LX-reuse facility is installed and is
enabled by a one value of the ASN-and-LX-reuse
control, bit 44 of control register 0, the secondary
ASTEIN (SASTEIN), bits 0-31 of control register 3,
is set equal to the primary ASTEIN (PASTEIN),
bits 0-31 of control register 4.

If the DASTE specifies the base space, the
subspace-active bit, bit 0 of word 1 of the DUCT,
is set to zero, and bits 1-31 of word 1 remain
unchanged. If the DASTE specifies a subspace
by means of ALET 1, then (1) the subspace-active
bit is set to one, (2) the SSASTEO in bit positions
1-25 of word 1 remains unchanged, and (3) bits
26-31 of word 1 either are set to zeros or remain
unchanged. If the DASTE specifies a subspace
by means of an ALET other than ALET 1, then (1)
the subspace-active bit is set to one, (2) the
DASTE is stored in bit positions 1-25 of word 1
as the SSASTEO, (3) zeros are stored in bit posi-
tions 26-31 of word 1, and (4) the ASTESN in the
DASTE is stored in word 3 of the DUCT as the
SSASTESN.

The fetch, store, and update references to the
DUCT are single-access references and appear to
be word concurrent as observed by other CPUs.
The words of the DUCT are accessed in no partic-
ular order.

The operation, since it changes a translation
parameter in control register 1, causes all copies
of prefetched instructions to be discarded, except
when in the home-space mode.

Special Conditions

DAT must be on; otherwise, a special-operation
exception is recognized. A special-operation
exception is also recognized if the current primary
address space is not in a subspace group associ-
ated with the current dispatchable unit, if the ALET
in access register R2 is ALET 1 but a subspace
has not previously been entered by the
dispatchable unit by means of BRANCH IN SUB-
SPACE GROUP, or if the ALET used is other than
ALET 0 and ALET 1 and the destination ASTE
does not specify the base space or a subspace of
the subspace group.

The primary space-switch-event-control bit, bit 57
of control register 1 either before or after the oper-
ation, does not cause a space-switch-event
program interruption to occur.

Key-controlled protection does not apply to any
access made during the operation. Low-address
protection does apply.

The operation is suppressed on all addressing and
protection exceptions.

The priority of recognition of program exceptions
for the instruction is shown in the figure "Priority of
Execution: BRANCH IN SUBSPACE GROUP."

Condition Code: The code remains unchanged.

Program Exceptions:

- Addressing (dispatchable-unit control table,
effective access-list designation, access-list
table, destination ASN-second-table entry)
- ALET specification
- ALEN translation
- ASTE sequence
- ASTE validity
- Protection (low-address; dispatchable-unit
control table)
- Special operation
- Trace
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Special-operation exception due to DAT being off.

8.A Trace exceptions.

8.B Protection exception (low-address protection) for access to dispatchable-unit control table.

8.C.1 Addressing exception for access to dispatchable-unit control table.

8.C.2 Special-operation exception due to current primary address space not being in a subspace group associated with the current dispatchable unit (primary-ASTE origin in control register 5 not equal to base-ASTE origin in dispatchable-unit control table).

**Note:** Exception 8.C.3.A can occur only if the access-list-entry token (ALET) in access register R2 is ALET 0.

8.C.3.A Addressing exception for access to base ASTE (ASTE designated by base-ASTE origin in dispatchable-unit control table).

**Note:** Exceptions 8.C.3.B.1-8.C.3.B.4 can occur only if the access-list-entry token (ALET) in access register R2 is ALET 1.

8.C.3.B.1 Special-operation exception due to subspace-ASTE origin in dispatchable-unit control table being zero.

8.C.3.B.2 Addressing exception for access to subspace ASTE.

8.C.3.B.3 ASTE-validity exception due to bit 0 in subspace ASTE being one.

8.C.3.B.4 ASTE-sequence exception due to ASTE sequence number in subspace ASTE not being equal to subspace-ASTE sequence number in dispatchable-unit control table.

**Note:** Exceptions 8.C.3.C.1-8.C.3.C.9 can occur only if the access-list-entry token (ALET) in access register R2 is other than ALET 0 and ALET 1.

8.C.3.C.1 ALET-specification exception due to bits 0-6 of ALET not being all zeros.

8.C.3.C.2 Addressing exception for access to effective access-list designation.

8.C.3.C.3 ALEN-translation exception due to access-list entry being outside the list.

**Figure 10-4 (Part 1 of 2). Priority of Execution: BRANCH IN SUBSPACE GROUP**
8.C.3.C.4 Addressing exception for access to access-list entry.

8.C.3.C.5 ALEN-translation exception due to I bit in access-list entry being one.

8.C.3.C.6 Addressing exception for access to destination ASTE.

8.C.3.C.7 ASTE-validity exception due to bit 0 in destination ASTE being one.

8.C.3.C.8 ASTE-sequence exception due to ASTE sequence number (ASTESN) in access-list entry not being equal to ASTESN in destination ASTE.

8.C.3.C.9 Special-operation exception due to destination-ASTE origin not equal to base-ASTE origin in dispatchable-unit control table and (1) subspace-group bit, bit 54 in address-space-control element in destination ASTE being zero or (2) base-space bit bit 31, in destination ASTE being one.

Figure 10-4 (Part 2 of 2). Priority of Execution: BRANCH IN SUBSPACE GROUP

Programming Notes:

1. See the discussion of BRANCH IN SUBSPACE GROUP in [Subroutine Linkage without the Linkage Stack] on page 5-11. It is intended that there be a separate ASN-second-table entry (ASTE) for each of the base space and each subspace of a subspace group. The ASTEs for the subspaces can be “pseudo” ASTEs as described in the programming note in [Address-Space Number] on page 3-17. A subspace can contain a subset of the storage in the base space by having the DAT tables for the subspace designate a subset of the pages that are designated by the DAT tables for the base space. A dispatchable unit has access to a subspace if an access-list entry designating the ASTE for the subspace is in the primary-space or dispatchable-unit access list of the dispatchable unit.

2. BRANCH IN SUBSPACE GROUP can be used to give control from the base space to a subspace, from a subspace to another subspace, and from a subspace to the base space. The instruction can also be used to give control from the base space to the base space or from a subspace to the same subspace.

3. Since BRANCH IN SUBSPACE GROUP sets the secondary address-space-control element (ASCE) in control register 7 equal to the new primary ASCE in control register 1 (along with setting the secondary ASN in control register 3 equal to the primary ASN in control register 4), the program in an address space given control by BRANCH IN SUBSPACE GROUP does not have access to the calling program’s address space by means of that address space being the secondary address space.

4. When a dispatchable unit has used BRANCH IN SUBSPACE GROUP to enter a subspace and has not subsequently used BRANCH IN SUBSPACE GROUP to return to the base space, the dispatchable unit is said to be “subspace active.” When LOAD ADDRESS SPACE PARAMETERS, PROGRAM CALL, PROGRAM RETURN, PROGRAM TRANSFER, PROGRAM TRANSFER WITH INSTANCE, SET SECONDARY ASN, or SET SECONDARY ASN WITH INSTANCE places an ASCE in control register 1 as the primary ASCE or in control register 7 as the secondary ASCE, and if (1) the ASCE has the subspace-group bit on in it, (2) the dispatchable unit is subspace active, and (3) the ASCE was obtained from the ASN-second-table entry (ASTE) for the base space of the current dispatchable unit, then the instruction (any of the seven named instructions) replaces bits 0-55 and 58-63 of the ASCE in the control register with the same bits of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. Further details about the effects of the subspace-group facility on the seven
named instructions are given in "Subspace-Replacement Operations" on page 5-68 and in the definitions of the instructions.

5. The use of BRANCH IN SUBSPACE GROUP (BSG) along with PROGRAM CALL (PC) and either PROGRAM TRANSFER (PT) (or PROGRAM TRANSFER WITH INSTANCE) or PROGRAM RETURN (PR) can produce results that may be unexpected. Consider the following sequence of operations:
   a. Start in the base space
   b. BSG to a subspace
   c. PC (the first PC) to an address space that is not in the subspace group.
   d. PC (the second PC) to the base space. Since the dispatchable unit is subspace active, control is given to the subspace.
   e. BSG back to the base space.
   f. PT or PR (paired with the second PC) back to the address space that is not in the subspace group.
   g. PT or PR (paired with the first PC) back to the subspace group. Since the dispatchable unit is no longer subspace active, control is given to the base space even though the first PC was issued in the subspace.

6. BRANCH IN SUBSPACE GROUP does not perform the serialization or checkpoint-synchronization functions, but it does cause all copies of prefetched instructions to be discarded except when in the home-space mode.

7. When the Rz field designates access register 0, the access register is treated as containing ALET 0 regardless of the contents of the access register.

**COMPARE AND SWAP AND PURGE**

CSP R1,R2 [RRE]

<table>
<thead>
<tr>
<th>'B250'</th>
<th>/ / / / /</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The first and second operands are compared. If they are equal, contents of general register R1 + 1 are stored at the second-operand location, and a purging operation is performed. If they are unequal, the second operand is loaded at the first-operand location. The result of the comparison is indicated in the condition code.

For COMPARE AND SWAP AND PURGE (CSP), the first operand is the contents of bit positions 32-63 of general register R1. The second operand is a word in storage.

For COMPARE AND SWAP AND PURGE (CSPG), the first operand is the contents of bit positions 0-63 of general register R1. The second operand is a doubleword in storage.

For both CSP and CSPG, the location of the left-most byte of the second operand is designated by contents of general register Rz.

The purging operation applies to ART-lookaside buffers (ALBs) and translation-lookaside buffers (TLBs) in all CPUs in the configuration. Either ALBs or TLBs, or both ALBs and TLBs, may be selected for purging. All entries are cleared from the selected buffers.

The purging operation is specified by means of bits 62 and 63 of general register Rz. When bit 62 is one, entries are cleared from ALBs. When bit 63 is one, entries are cleared from TLBs. When bits 62 and 63 both are ones, entries are cleared from ALBs and TLBs. When bits 62 and 63 both are zeros, no entries are cleared.

The handling of the address in general register Rz is dependent on the addressing mode. For CSP in the 24-bit addressing mode, the contents of bit positions 40-61 of general register Rz, with two zeros appended on the right, constitute the address, and the contents of bit positions 0-39 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-61 of the register, with two zeros appended on the right, constitute the address, and the contents of bit positions 0-39 are ignored. In the 64-bit addressing mode, the con-
tents of bit positions 0-61 of the register, with two zeros appended on the right, constitute the address.

For CSPG in the 24-bit addressing mode, the contents of bit positions 40-60 of general register R, with three zeros appended on the right, constitute the address, and the contents of bit positions 0-39 and 61 are ignored. In the 31-bit addressing mode, the contents of bit positions 33-60 of the register, with three zeros appended on the right, constitute the address, and the contents of bit positions 0-32 and 61 are ignored. In the 64-bit addressing mode, the contents of bit positions 0-60 of the register, with three zeros appended on the right, constitute the address, and the contents of bit position 61 are ignored.

The contents of the registers just described are shown in Figure 10-5 on page 10-20. When an equal comparison occurs, the contents of bit positions 32-63 of general register R₁ + 1 for CSP, or of bit positions 0-63 for CSPG, are stored at the second-operand location. The fetch of the second operand for purposes of comparison and the store into the second-operand location appear to be a block-concurrent interlocked-update reference as observed by other CPUs.

When the result of the comparison is unequal, the second-operand is loaded at the first-operand location, bits 0-31 of general register R₁ remain unchanged for CSP only, and the second-operand location remains unchanged. However, on some models, the second operand may be fetched and subsequently stored back unchanged at the second-operand location. This update appears to be a block-concurrent interlocked-update reference as observed by other CPUs.

A serialization function is performed before the operand is fetched and again after the operation is completed.

When an equal comparison occurs, this CPU clears entries from its ALB and TLB, as specified by bits 62 and 63 of general register R₂, and signals all CPUs in the configuration to clear the same specified entries from their ALBs and TLBs. The ALB entries that are cleared are all ALB access-list designations, access-list entries, ASN-second-table entries, and authority-table entries. The TLB entries that are cleared are all combined region-and-segment-table entries, page-table entries, and real-space entries.

The execution of COMPARE AND SWAP AND PURGE is not completed on the CPU which executes it until (1) all specified entries have been cleared from the ALB and TLB of this CPU and (2) all other CPUs in the configuration have completed any storage accesses, including the updating of the change and reference bits, by using the specified ALB and TLB entries.

Special Conditions

The R₁ field must designate an even register; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0 First and second operands equal, second operand replaced by contents of general register R₁ + 1
1 First and second operands unequal, first operand replaced by second operand
2 --
3 --

**Program Exceptions:**

- Access (fetch and store, operand 2)
- Operation (if DAT-enhancement facility is not installed, CSPG only)
- Privileged operation
- Specification

**Programming Note:** COMPARE AND SWAP AND PURGE provides a broadcast form of the PURGE ALB and PURGE TLB instructions, thus making it possible to avoid uses of SIGNAL PROCESSOR.

**DIAGNOSE**

<table>
<thead>
<tr>
<th>'83'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 8 31</td>
</tr>
</tbody>
</table>

The CPU performs built-in diagnostic functions, or other model-dependent functions. The purpose of the diagnostic functions is to verify proper functioning of equipment and to locate faulty components. Other model-dependent functions may include disabling of failing buffers, reconfiguration of CPUs, storage, and channel paths, and modification of control storage.

Chapter 10. Control Instructions 10-19
Bits 8-31 may be used as in the SI or RS formats, or in some other way, to specify the particular diagnostic function. The use depends on the model.

The execution of the instruction may affect the state of the CPU and the contents of a register or storage location, as well as the progress of an I/O operation. Some diagnostic functions may cause the test indicator to be turned on.

**Resulting Condition Code:** The code is unpredictable.

**Program Exceptions:**
- Privileged operation
- Depending on the model, other exceptions may be recognized.

**Programming Notes:**
1. Since the instruction is not intended for problem-state-program or control-program use, DIAGNOSE has no mnemonic.

2. DIAGNOSE, unlike other instructions, does not follow the rule that programming errors are distinguished from equipment errors. Improper use of DIAGNOSE may result in false machine-check indications or may cause actual machine malfunctions to be ignored. It may also alter other aspects of system operation, including instruction execution and channel-program operation, to an extent that the operation does not comply with that specified in this publication. As a result of the improper use of DIAGNOSE, the system may be left in such a condition that the power-on reset or initial-microprogram-loading (IML) function must be performed. Since the function performed by DIAGNOSE may differ from model to model and between versions of a model, the program should avoid issuing
DIAGNOSE unless the program recognizes both the model number and version code stored by STORE CPU ID.

EXTRACT AND SET EXTENDED AUTHORITY

EXTRACT PRIMARY ASN

EXTRACT PRIMARY ASN AND INSTANCE
EXTRACT SECONDARY ASN

ESAR  R₁  [RRE]

<table>
<thead>
<tr>
<th>'B227'</th>
<th>////////</th>
<th>R₁</th>
<th>/////</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

EXTRACT SECONDARY ASN AND INSTANCE

ESAIR  R₁  [RRE]

<table>
<thead>
<tr>
<th>'B99B'</th>
<th>////////</th>
<th>R₁</th>
<th>/////</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The 16-bit SASN, bits 48-63 of control register 3, is placed in bit positions 48-63 of general register R₁. Bits 32-47 of the general register are set to zeros.

In the EXTRACT SECONDARY ASN AND INSTANCE operation, the SASTEIN, bits 0-31 of control register 3, is placed in bit positions 0-31 of general register R₁. In the EXTRACT SECONDARY ASN operation, bits 0-31 of general register R₁ remain unchanged.

Special Conditions

The instruction must be executed with DAT on; otherwise, a special-operation exception is recognized.

In the problem state, the extraction-authority control, bit 36 of control register 0, must be one; otherwise, a privileged-operation exception is recognized. In the supervisor state, the extraction-authority-control bit is not examined.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-7.

Condition Code: The code remains unchanged.

Program Exceptions:

- Operation (if the ASN-and-LX-reuse facility is not installed, ESAIR only)
- Privileged operation (extraction-authority control is zero in the problem state)
- Special operation

1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.
7.B.1 Operation exception if the ASN-and-LX-reuse facility is not installed (ESAIR only).
7.B.2 Special-operation exception due to DAT being off.
8. Privileged-operation exception due to extraction-authority control, bit 36 of control register 0, being zero in problem state.

Figure 10-7. Priority of Execution: EXTRACT SECONDARY ASN

EXTRACT STACKED REGISTERS

EREG  R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>'B249'</th>
<th>////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

EREGG  R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>'B90E'</th>
<th>////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

Contents of a set of general registers and a set of access registers that were saved in the last state entry in the linkage stack are restored to the registers. Each set of registers begins with register R₁ and ends with register R₂.

For EXTRACT STACKED REGISTERS (EREG), the contents of bit positions 32-63 of the general registers are restored, and the contents of bit positions 0-31 of the registers remain unchanged. For EXTRACT STACKED REGISTERS (EREGG), the contents of bit positions 0-63 of the general registers are restored. In either case, the contents of bit positions 0-31 of the access registers are restored.

For each of the general registers and the access registers, the registers are loaded in ascending order of their register numbers, starting with reg-
ister R₁ and continuing up to and including register R₁₅, with register 0 following register 15. The bit positions of each register are loaded from the position in the state entry where the contents of the bit positions were saved when the state entry was created. The contents of the state entry remain unchanged.

The last state entry is located as described in “Unstacking Process” on page 5-86. The state entry remains in the linkage stack, and the linkage-stack-entry address in control register 15 remains unchanged.

Key-controlled protection does not apply to references to the linkage stack.

**Special Conditions**

The CPU must be in the primary-space mode, access-register mode, or home-space mode; otherwise, a special-operation exception is recognized.

A stack-empty, stack-specification, or stack-type exception may be recognized during the unstacking process.

The operation is suppressed on all addressing exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-8 on page 10-24.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, except for protection, linkage-stack entry)
- Special operation
- Stack empty
- Stack specification
- Stack type
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Special-operation exception due to DAT being off or the CPU being in secondary-space mode.

8. Access exceptions (fetch) for entry descriptor of the current linkage-stack entry.

9. Stack-type exception due to current entry not being a state entry or header entry.

Note: Exceptions 10-14 can occur only if the current entry is a header entry.

10. Access exceptions (fetch) for second word of the header entry.

11. Stack-empty exception due to backward stack-entry validity bit in the header entry being zero.

12. Access exceptions (fetch) for entry descriptor of preceding entry, which is the entry designated by the backward stack-entry address in the current (header) entry.

13. Stack-specification exception due to preceding entry being a header entry.

14. Stack-type exception due to preceding entry not being a state entry.

15. Access exceptions (fetch) for the selected contents of the state entry.

Note: Exceptions 1/zerodot-14 can occur only if the current entry is a header entry.

1/zerodot. Access exceptions (fetch) for second instruction halfword.

11. Stack-empty exception due to backward stack-entry validity bit in the header entry being zero.

12. Access exceptions (fetch) for entry descriptor of preceding entry, which is the entry designated by the backward stack-entry address in the current (header) entry.

13. Stack-specification exception due to preceding entry being a header entry.

14. Stack-type exception due to preceding entry not being a state entry.

15. Access exceptions (fetch) for the selected contents of the state entry.

Figure 10-8. Priority of Execution: EXTRACT STACKED REGISTERS

EXTRACT STACKED STATE

ESTA R1,R2 [RRE]

<table>
<thead>
<tr>
<th>'B24A'</th>
<th>/ / / / / /</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sixty-four or 128 bits of status information in the last state entry in the linkage stack are placed in the pair of general registers designated by the R1 field. The condition code is set to indicate whether the state entry is a branch state entry or a program-call state entry.

The R1 field designates the even-numbered register of an even-odd pair of general registers.

Bits 56-63 of general register R2 are an unsigned binary integer that is used as a code to select the state-entry byte positions, or byte and bit positions, from which information is to be extracted, as follows:

<table>
<thead>
<tr>
<th>Code (Bits 56-63 of Gen. Reg. R2)</th>
<th>State-Entry Byte Positions, or Byte and Bit Positions, Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128-135</td>
</tr>
<tr>
<td>1</td>
<td>136-139, 140.0, and 168-175.33-63 (see text)</td>
</tr>
<tr>
<td>2</td>
<td>144-151</td>
</tr>
<tr>
<td>3</td>
<td>152-159</td>
</tr>
<tr>
<td>4</td>
<td>136-143 and 168-175</td>
</tr>
<tr>
<td>5</td>
<td>176-183</td>
</tr>
</tbody>
</table>

For a code of 0, 2, or 3 in bit positions 56-63 of general register R2, the contents of the leftmost four bytes of the eight bytes of status information are placed in bit positions 32-63 of general register R1, and the contents of the rightmost four bytes of the status information are placed in bit positions 32-63 of general register R1 + 1. The
contents of bit positions 0-31 of general registers \( R_1 \) and \( R_1 + 1 \) remain unchanged.

For a code of 1 in bit positions 56-63 of general register \( R_2 \), the contents of bytes 136-139 of the state entry, which are bits 0-31 of the PSW in the state entry, are placed in bit positions 32-63 of general register \( R_1 \); the contents of bit position 0 of byte 140 of the entry, which is bit 32 of that PSW, are placed in bit position 32 of general register \( R_1 + 1 \); and the contents of bit positions 33-63 of bytes 168-175 of the entry, which are bits 97-127 of the PSW, are placed in bit positions 33-63 of general register \( R_1 + 1 \). However, bit 44 of general register \( R_1 \), which corresponds to bit 12 of the PSW in the state entry, is set to one, indicating the ESA/390 mode. Also, if bits 0-32 of bytes 168-175 of the state entry are not all zeros, bit 63 of general register \( R_1 + 1 \) is set to one; otherwise, bit 63 remains with the value loaded from bit position 63 of bytes 168-175 of the state entry. The contents of bit positions 0-31 of general registers \( R_1 \) and \( R_1 + 1 \) remain unchanged.

For a code of 4 in bit positions 56-63 of general register \( R_2 \), the contents of bytes 136-143 of the state entry, which are bits 0-63 of the PSW in the state entry, are placed in bit positions 0-63 of general register \( R_1 \), and the contents of bytes 168-175 of the state entry, which are bits 64-127 of that PSW, are placed in bit positions 0-63 of general register \( R_1 + 1 \).

Code 5 in bit positions 56-63 of general register \( R_2 \) is valid only if the ASN-and-LX-reuse facility is installed. For code 5, the contents of the leftmost four bytes of the eight bytes of status information are placed in bit positions 0-31 of general register \( R_1 \), and the contents of the rightmost four bytes of the status information are placed in bit positions 0-31 of general register \( R_1 + 1 \). The contents of bit positions 32-63 of general registers \( R_1 \) and \( R_1 + 1 \) remain unchanged.

The format of byte positions 128-183 of the state entry is as follows:

<table>
<thead>
<tr>
<th>PKM</th>
<th>SASN</th>
<th>EAX</th>
<th>PASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>130</td>
<td>132</td>
<td>134</td>
</tr>
</tbody>
</table>

If ASN-and-LX Reuse Is Enabled; otherwise Unpredictable

<table>
<thead>
<tr>
<th>SASTEIN</th>
<th>PASTEIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>180</td>
</tr>
</tbody>
</table>

Chapter 10. Control Instructions 10-25
The contents of the state entry remain unchanged.

The last state entry is located as described in "Unstacking Process" on page 5-86. The state entry remains in the linkage stack, and the linkage-stack-entry address in control register 15 remains unchanged.

When the entry-type code in the entry descriptor of the state entry is 0001100 binary, indicating a branch state entry, the condition code is set to 0. When the entry-type code is 0001101 binary, indicating a program-call state entry, the condition code is set to 1.

Key-controlled protection does not apply to references to the linkage stack.

Bits 0-55 of general register R₂ are ignored.

**Special Conditions**

A specification exception is recognized when R₁ is odd or the code in bit positions 56-63 of general register R₂ is greater than 4 when the ASN-and-LX-reuse facility is not installed or is greater than 5 when the facility is installed.

The CPU must be in the primary-space mode, access-register mode, or home-space mode; otherwise, a special-operation exception is recognized.

A stack-empty, stack-specification, or stack-type exception may be recognized during the unstacking process.

The operation is suppressed on all addressing exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-9 on page 10-27.

**Resulting Condition Code:**

0  Branch state entry
1  Program-call state entry
2  --
3  --

**Program Exceptions:**

- Access (fetch, except for protection, linkage-stack entry)
- Special operation
- Specification
- Stack empty
- Stack specification
- Stack type

**Programming Note:** The results for a code of 1 in bit positions 56-63 of general register R₂ are intended to provide compatibility with ESA/390. (It may be that only values of bits in bit positions 0-31 of the PSW are required.) Bit 63 of general register R₁ + 1 is set to one if the instruction address in the PSW in the state entry is larger than a 31-bit address.
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Special-operation exception due to DAT being off or the CPU being in secondary-space mode.

8.A Specification exception due to $R_1$ being odd or bits 56-63 of general register $R_2$ having a value greater than 4 when the ASN-and-LX-reuse facility is not installed or greater than 5 when the facility is installed.

8.B.1 Access exceptions (fetch) for entry descriptor of the current linkage-stack entry.

8.B.2 Stack-type exception due to current entry not being a state entry or header entry.

8.B.3 Access exceptions (fetch) for second word of the header entry.

8.B.4 Stack-empty exception due to backward stack-entry validity bit in the header entry being zero.

8.B.5 Access exceptions (fetch) for entry descriptor of preceding entry, which is the entry designated by the backward stack-entry address in the current (header) entry.

8.B.6 Stack-specification exception due to preceding entry being a header entry.

8.B.7 Stack-type exception due to preceding entry not being a state entry.

8.B.8 Access exceptions (fetch) for the selected contents of the state entry.

---

<table>
<thead>
<tr>
<th>IAC</th>
<th>R1</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B224'</td>
<td>///</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10-9. Priority of Execution: EXTRACT STACKED STATE

INSERT ADDRESS SPACE CONTROL

The address-space-control bits, bits 16 and 17 of the current PSW, are placed in reversed order in bit positions 54 and 55 of general register $R_1$; that is, bit 16 is placed in bit position 55, and bit 17 is placed in bit position 54. Bits 48-53 of the register are set to zeros, and bits 0-47 and 56-63 of the register remain unchanged. The address-space-control bits are also used to set the condition code.

**Special Conditions**

The instruction must be executed with DAT on; otherwise, a special-operation exception is recognized.

In the problem state, the extraction-authority control, bit 36 of control register 0, must be one; otherwise, a privileged-operation exception is recognized. In the supervisor state, the extraction-authority-control bit is not examined.
The priority of recognition of program exceptions for the instruction is shown in Figure 10-10 on page 10-28.

**Resulting Condition Code:**
- 0  PSW bits 16 and 17 zeros (indicating primary-space mode)
- 1  PSW bit 16 one and bit 17 zero (indicating secondary-space mode)
- 2  PSW bit 16 zero and bit 17 one (indicating access-register mode)
- 3  PSW bits 16 and 17 ones (indicating home-space mode)

**Program Exceptions:**
- Privileged operation (extraction-authority control is zero in the problem state)
- Special operation

1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Special-operation exception due to DAT being off.

8. Privileged-operation exception due to extraction-authority control, bit 36 of control register 0, being zero in problem state.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Privileged operation (extraction-authority control is zero in the problem state)

**INSERT STORAGE KEY EXTENDED**

The storage key for the block that is addressed by the contents of general register $R_2$ is inserted in general register $R_1$.

In the 24-bit addressing mode, bits 40-51 of general register $R_2$ designate a 4K-byte block in real storage, and bits 0-39 and 52-63 of the register are ignored. In the 31-bit addressing mode, bits 33-51 of general register $R_2$ designate a 4K-byte block in real storage, and bits 0-32 and 52-63 of the register are ignored. In the 64-bit addressing mode, bits 0-51 of general register $R_2$ designate a 4K-byte block in real storage, and bits 52-63 of the register are ignored.
The address designating the storage block, being a real address, is not subject to dynamic address translation. The reference to the storage key is not subject to a protection exception.

The seven-bit storage key is inserted in bit positions 56-62 of general register \( R_1 \), and bit 63 is set to zero. The contents of bit positions 0-55 of the register remain unchanged.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Addressing (address specified by general register \( R_2 \))
- Privileged operation

### INSERT VIRTUAL STORAGE KEY

**IVSK \( R_1, R_2 \) [RRE]**

<table>
<thead>
<tr>
<th>'B223'</th>
<th>//////////</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

The storage key for the location designated by the virtual address in general register \( R_2 \) is inserted in general register \( R_1 \).

Selected bits of general register \( R_2 \) are used as a virtual address. In the 24-bit addressing mode, the address is specified by bits 40-63 of the register, and bits 0-39 are ignored. In the 31-bit addressing mode, the address is specified by bits 33-63, and bits 0-32 is ignored. In the 64-bit addressing mode, the address is specified by bits 0-63 of the register.

The address is a virtual address and is subject to the address-space-control bits, bits 16 and 17 of the current PSW. The address is treated as a primary virtual address in the primary-space mode, as a secondary virtual address in the secondary-space mode, as an AR-specified virtual address in the access-register mode, or as a home virtual address in the home-space mode. The reference to the storage key is not subject to a protection exception.

Bits 0-4 of the storage key, which are the access-control bits and the fetch-protection bit, are placed in bit positions 56-60 of general register \( R_1 \), with bits 61-63 set to zeros. The contents of bit positions 0-55 of the register remain unchanged. The change and reference bits in the storage key are not inspected. The change bit is not affected by the operation. The reference bit, depending on the model, may or may not be set to one as a result of the operation.

The following diagram shows the storage key and the register positions just described.

```
Storage Key for the Location

<table>
<thead>
<tr>
<th>ACC</th>
<th>F</th>
<th>R</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zeros</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

└───┬──┘
    │
    │ ACC | F | /zerodot/zerodot/zerodot |

\( R_1 \)

0 16 24 28 31
0 56 60 63

Zeros

Special Conditions

The instruction must be executed with DAT on; otherwise, a special-operation exception is recognized.

In the problem state, the extraction-authority control, bit 36 of control register 0, must be one; otherwise, a privileged-operation exception is recognized. In the supervisor state, the extraction-authority-control bit is not examined.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-11 on page 10-30.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (except for protection, address specified by general register \( R_2 \))
- Privileged operation (extraction-authority control is zero in the problem state)
- Special operation
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Special-operation exception due to DAT being off.

8. Privileged-operation exception due to extraction-authority control, bit 36 of control register /zerodot, being zero.

9. Access exceptions (except for protection) for address specified by general register R_2.

Figure 10-11. Priority of Execution: INSERT VIRTUAL STORAGE KEY

Programming Notes:

1. Since all bytes in a 4K-byte block are associated with the same page and the same storage key, bits 52-63 of general register R/superone/basesupertwo essentially are ignored.

2. In the access-register mode, access register 0 designates the primary address space regardless of the contents of access register 0.

INVALIDATE DAT TABLE ENTRY

IDTE  R_1,R_3,R_2  [RRF]

<table>
<thead>
<tr>
<th>'B98E'</th>
<th>R_3</th>
<th>///</th>
<th>R_1</th>
<th>R_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

When the clearing-by-ASCE-option bit, bit 52 of general register R_2, is zero, an operation called the invalidation-and-clearing operation is performed, as follows. The designated region-table entry or segment-table entry in storage, or a range of entries beginning with the designated entry, is invalidated, and the translation-lookaside buffers (TLBs) in all CPUs in the configuration are cleared of (1) all combined region-and-segment-table entries (CRSTEs) and page-table entries (PTEs) formed through use of the invalidated entry or entries in storage and (2) all PTEs formed through use of a CRSTE formed through use of the invalidated entry or entries in storage. The TLB entries cleared may optionally be limited to entries formed to translate addresses in a specified address space.

When the clearing-by-ASCE-option bit is one, an operation called the clearing-by-ASCE operation is performed, as follows. The operation does not perform any invalidation of DAT-table entries in storage, but it does clear, from the TLBs in all CPUs in the configuration, all combined region-and-segment-table entries and page-table entries formed to translate addresses in a specified address space.

The two operations are described separately below, before the section “Common Operation.”

Invalidation-and-Clearing Operation

When bit 52 of general register R_2, the clearing-by-ASCE-option bit, is zero, the invalidation-and-clearing operation is specified.

The contents of general register R_1 have the format of an address-space-control element with only the table origin, bits 0-51, and designation-type control (DT), bits 60 and 61, used. The table origin designates the DAT table in which one or more entries are to be invalidated, and DT specifies the type of that table.

Bits 52-59, 62, and 63 of general register R_1 are ignored.

Bits 0-43 of general register R_2 have the format of the region index and segment index of a virtual address. The part of bits 0-43 normally used by DAT to select an entry in the type of table designated by general register R_1 is called the effective invalidation index.

Bits 60 and 61 of general register R_1 specify a table type and an effective invalidation index as follows:
The part of bits 0-43 of general register R to the right of the effective invalidation index is ignored.

The table origin in general register R and the effective invalidation index designate a DAT-table entry to be invalidated. Bits 53-63 of general register R are an unsigned binary integer specifying the number of additional table entries to be invalidated. Therefore, the number of entries to be invalidated is 1-2,048, corresponding to a value of bits 53-63 of 0-2,047.

Bits 44-51 of general register R must be zeros; otherwise, a specification exception is recognized.

If R is nonzero, the contents of general register R have the format of an address-space-control element with only the table origin, bits 0-51, and designation-type control (DT), bits 60 and 61, used. These contents are used to select TLB entries to be cleared. Bits 52-59, 62, and 63 of general register R are ignored. If R is zero, the entire contents of general register 0 are ignored.

The contents of the general registers just described are shown in Figure 10-12.

The table origin in general register R and effective invalidation index in general register R designate a table entry in accordance with the rules in "Lookup in a Table Designated by an Address-Space-Control Element" on page 3-46, except that a carry from bit position 0 of the resulting address is always ignored, and the index is not checked against a table-length field. The table origin is treated as a 64-bit address, and the addition is performed by using the rules for 64-bit address arithmetic, regardless of the current addressing mode specified by bits 31 and 32 of the current PSW. The address formed from these two components is a real or absolute address. The invalid bit, bit 58, of this region-table or segment-table entry is set to one. During this procedure, the entry is not checked for a format error or whether the origin, in the entry, of the next-lower-level table would cause an addressing exception. The table-type field in the entry is ignored. If the entry is a segment-table entry, the common-segment bit in the entry is ignored.

### Figure 10-12. Register Contents for INVALIDATE DAT TABLE ENTRY Invalidation-and-Clearing Operation (Bit 52 of GR R Is Zero)

<table>
<thead>
<tr>
<th>R1</th>
<th>Table Type</th>
<th>Effective Invalidation Index in Reg. R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Region first table</td>
<td>Region first index (bits 0-10)</td>
</tr>
<tr>
<td>10</td>
<td>Region second table</td>
<td>Region second index (bits 11-21)</td>
</tr>
<tr>
<td>01</td>
<td>Region third table</td>
<td>Region third index (bits 22-32)</td>
</tr>
<tr>
<td>00</td>
<td>Segment table</td>
<td>Segment index (bits 33-43)</td>
</tr>
</tbody>
</table>

The part of bits 0-43 of general register R to the right of the effective invalidation index is ignored.

---

<table>
<thead>
<tr>
<th>R1</th>
<th>Region-Table or Segment-Table Origin for Invalidation</th>
<th>/ / / / / / /</th>
<th>DT</th>
<th>//</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>52</td>
<td>60</td>
<td>63</td>
</tr>
</tbody>
</table>

Note: The effective invalidation index is specified by DT in register R1.

<table>
<thead>
<tr>
<th>R2</th>
<th>Reg.-First Index</th>
<th>Reg.-Second Index</th>
<th>Reg.-Third Index</th>
<th>Segment Index</th>
<th>00000000</th>
<th>Additional Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>52</td>
<td>63</td>
</tr>
</tbody>
</table>

Note: Used only if R is nonzero.

<table>
<thead>
<tr>
<th>R3</th>
<th>Region-Table or Segment-Table Origin for Clearing</th>
<th>/ / / / / / /</th>
<th>DT</th>
<th>//</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>52</td>
<td>60</td>
<td>63</td>
</tr>
</tbody>
</table>
The entire table entry is fetched concurrently from storage. Subsequently, the byte containing the invalid bit is stored. The fetch access to the entry is subject to key-controlled protection, and the store access is subject to key-controlled protection and low-address protection.

If bits 53-63 of general register \( R_2 \) are not all zeros, the setting of the invalid bit to one in a region-table or segment-table entry is repeated by adding one to the previously used value of the effective invalidation index, and this is done as many times as are specified by bits 53-63. A carry out of the leftmost bit position of the effective invalidation index is ignored, and wraparound in the table occurs in this case. The contents of general register \( R_2 \) remain unchanged.

A serialization function is performed before the operation begins and again after the operation is completed. As is the case for all serialization operations, this serialization applies only to this CPU; other CPUs are not necessarily serialized.

After it has set an invalid bit to one, this CPU clears selected entries from its TLB and signals all other CPUs in the configuration to clear selected entries from their TLBs. Each TLB is cleared of at least those combined region-and-segment-table entries (CRSTEs) for which all of the following conditions are met:

- The effective invalidation index and all bits to the left of it in general register \( R_2 \) match the same part of the region-and-segment-index field in the CRSTE. Note that when multiple table entries are invalidated due to bits 53-63 of general register \( R_2 \), then the effective invalidation index is incremented, a carry out of the leftmost bit position of the index is lost, and CRSTEs are cleared for each value of the index so obtained.
- Either \( R_3 \) is zero or the table-origin and designation-type fields in general register \( R_3 \) match the table-origin and designation-type fields in the address-space-control element (ASCE) used to form the CRSTE. This ASCE may be one that attached a translation path containing a segment-table entry that attached the PTE in storage from which the TLB PTE was formed, or it may be one that made usable a CRSTE that attached the PTE in storage from which the TLB PTE was formed. See "Formation of TLB Entries" on page 3-50 for the meaning of the terminology used here.
- If the entry invalidated in storage is a segment-table entry, the page-table-origin field in the invalidated entry matches the page-table-origin field in the CRSTE.

Each TLB is also cleared of at least those page-table entries (PTEs) for which all of the following conditions are met:

- The TLB PTE was formed through use of an entry invalidated in storage or through use of a CRSTE formed through use of an entry invalidated in storage.
- Either \( R_3 \) is zero or the table-origin and designation-type fields in general register \( R_3 \) match the table-origin and designation-type fields in the address-space-control element (ASCE) used to form the TLB PTE. This ASCE may be one that attached a translation path containing a segment-table entry that attached the PTE in storage from which the TLB PTE was formed, or it may be one that made usable a CRSTE that attached the PTE in storage from which the TLB PTE was formed. See "Formation of TLB Entries" on page 3-50 for the meaning of the terminology used here.
- If the entry invalidated in storage is a segment-table entry, the page-table-origin field in the invalidated entry matches the page-table-origin field in the TLB PTE.

Storing in the region- or segment-table entry and the clearing of TLB entries may or may not occur if the invalid bit is already one in the region- or segment-table entry.

When multiple entries are invalidated, clearing of TLB entries may be delayed until all entries have been invalidated.

**Clearing-by-ASCE Operation**

When bit 52 of general register \( R_2 \), the clearing-by-ASCE-option bit, is one, the clearing-by-ASCE operation is specified.

The contents of general register \( R_3 \) have the format of an address-space-control element with only the table origin, bits 0-51, and designation-type control (DT), bits 60 and 61, used. These contents are used to select TLB entries to be cleared. Bits 52-59, 62, and 63 of general register \( R_3 \) are ignored. \( R_3 \) may be zero or nonzero, that
is, any general register, including register 0, may be designated.

Bits 44-51 of general register \( R_2 \) must be zeros; otherwise, a specification exception is recognized.

The contents of general register \( R_1 \) and of bit positions 0-43 and 53-63 of general register \( R_2 \) are ignored.

The contents of the general registers just described are shown in Figure 10-13.

The TLBs of all CPUs in the configuration are cleared of at least those combined region-and-segment-table entries (CRSTEs) for which the table-origin and designation-type fields in general register \( R_3 \) match the table-origin and designation-type fields in the address-space-control element (ASCE) used to form the CRSTE. This ASCE is the one that attached the translation path from which the CRSTE was formed. See “Formation of TLB Entries” on page 3-50 for the meaning of the terminology used here.

Each TLB is also cleared of at least those page-table entries (PTEs) for which the table-origin and designation-type fields in general register \( R_3 \) match the table-origin and designation-type fields in the address-space-control element (ASCE) used to form the TLB PTE. This ASCE may be one that attached a translation path containing a segment-table entry that attached the PTE in storage from which the TLB PTE was formed, or it may be one that made usable a CRSTE that attached the PTE in storage from which the TLB PTE was formed. See “Formation of TLB Entries” on page 3-50 for the meaning of the terminology used here.

Common Operation

When the clearing-by-ASCE-option bit, bit 52 of general register \( R_2 \), is either zero or one, the execution of INVALIDATE DAT TABLE ENTRY is not completed on the CPU which executes it until (1) all entries corresponding to the specified parameters have been cleared from the TLB of this CPU and (2) all other CPUs in the configuration have completed any storage accesses, including the updating of the change and reference bits, by using TLB entries corresponding to the specified parameters.

The operations do not necessarily have any effect on TLB real-space entries.

Special Conditions

Bits 44-51 of general register \( R_2 \) must be zeros; otherwise, a specification exception is recognized.

The operation is suppressed on all addressing and protection exceptions (invalidation-and-clearing operation only).

**Resulting Condition Code:** The code is unpredictable.

---

**Figure 10-13. Register Contents for INVALIDATE DAT TABLE ENTRY Clearing-by-ASCE Operation (Bit 52 of GR \( R_2 \) Is One)**

<table>
<thead>
<tr>
<th>( R_1 )</th>
<th>( R_2 )</th>
<th>( R_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>/////////////////////////////////////////////////////////////////////</td>
<td>00000000 1 /////////////</td>
<td>Region-Table or Segment-Table Origin for Clearing ////////// DT //</td>
</tr>
</tbody>
</table>

Note: Used if \( R_3 \) is zero or nonzero.
Program Exceptions:

- Addressing (invalidated region- or segment-table entry, invalidation-and-clearing operation only).
- Privileged operation
- Operation (if the DAT-enhancement facility is not installed)
- Protection (fetch and store, region- or segment-table entry, key-controlled protection and low-address protection; invalidation-and-clearing operation only).

Programming Notes:

1. The selective clearing of TLB entries may be implemented in different ways, depending on the model, and, in general, more entries may be cleared than the minimum number required. When the invalidation-and-clearing operation is performed, some models may clear all TLB entries when the effective invalidation index is not a segment index or may clear an entry regardless of the page-table origin in the entry. When that operation or the clearing-by-ASCE operation is performed, some models may clear a TLB entry regardless of the designation-type field in general register R\textsubscript{3}. When either operation is performed, other models may clear precisely the minimum number of entries required. Therefore, in order for a program to operate on all models, the program should not take advantage of any properties obtained by a less selective clearing on a particular model.

2. When using the clearing-by-ASCE operation to clear TLB entries associated with common segments, note that these entries may have been formed through use of address-space-control elements containing many different table origins.

The following notes apply when the invalidation-and-clearing operation is specified.

3. The clearing of TLB entries may make use of the page-table origin in a segment-table entry. Therefore, if the segment-table entry, when in the attached state, ever contained a page-table origin that is different from the current value, copies of entries containing the previous values may remain in the TLB.

4. INVALIDATE DAT TABLE ENTRY cannot be safely used to update a shared location in main storage if the possibility exists that another CPU or a channel program may also be updating the location.

5. The address of the DAT-table entry for INVALIDATE DAT TABLE ENTRY is a 64-bit address, and the address arithmetic is performed by following the normal rules for 64-bit address arithmetic, with wraparound at \(2^{64} - 1\). Also, offset and length fields are not used. Contrast this with implicit translation and the translations for LOAD REAL ADDRESS and STORE REAL ADDRESS, all of which may result either in wraparound or in an addressing exception when a carry occurs out of bit position 0 and which indicate an exception condition when the designated entry does not lie within its table. Accordingly, the DAT tables should not be specified to wrap from maximum storage locations to location 0, and the first designated entry and all additional entries specified by bits 53-63 of general register 3 should lie within the designated table.

**INVALIDATE PAGE TABLE ENTRY**

\[
\text{IPTE } \text{R}_1, \text{R}_2 \quad \text{[RRE]}
\]

\[
\begin{array}{cccc}
\text{‘B221’} & \\
0 & 16 & 24 & 28 & 31
\end{array}
\]

The designated page-table entry is invalidated, and the translation-lookaside buffers (TLBs) in all CPUs in the configuration are cleared of the associated entries.

The contents of general register \(R_1\) have the format of a segment-table entry, with only the page-table origin used. The contents of general register \(R_2\) have the format of a virtual address, with only the page index used. The contents of fields that are not part of the page-table origin or page index are ignored.

The contents of the general registers just described are as follows:

\[
\begin{array}{cccc}
\text{Page-Table Origin} & \\
0 & 53 & 63
\end{array}
\]
The page-table origin and the page index designate a page-table entry, following the dynamic-address-translation rules for page-table lookup. The page-table origin is treated as a 64-bit address, and the addition is performed by using the rules for 64-bit address arithmetic, regardless of the current addressing mode, which is specified by bits 31 and 32 of the current PSW. A carry out of bit position 0 as a result of the addition of the page index and page-table origin cannot occur. The address formed from these two components is a real or absolute address. The page-invalid bit of this page-table entry is set to one. During this procedure, the page-table entry is not inspected for whether the page-invalid bit is already one or for format errors. Additionally, the page-frame real address contained in the entry is not checked for an addressing exception.

The entire page-table entry appears to be fetched concurrently from storage as observed by other CPUs. Subsequently, the byte containing the page-invalid bit is stored. The fetch access to the page-table entry is subject to key-controlled protection, and the store access is subject to key-controlled protection and low-address protection.

A serialization function is performed before the operation begins and again after the operation is completed. As is the case for all serialization operations, this serialization applies only to this CPU; other CPUs are not necessarily serialized.

If no exceptions are recognized, this CPU clears selected entries from its TLB and signals all CPUs in the configuration to clear selected entries from their TLBs. Each TLB is cleared of at least those entries that have been formed using all of the following:

- The page-table origin specified by general register $R_1$
- The page index specified by general register $R_2$
- The page-frame real address contained in the designated page-table entry

The execution of INVALIDATE PAGE TABLE ENTRY is not completed on the CPU which executes it until (1) all entries corresponding to the specified parameters have been cleared from the TLB of this CPU and (2) all other CPUs in the configuration have completed any storage accesses, including the updating of the change and reference bits, by using TLB entries corresponding to the specified parameters.

**Special Conditions**

The operation is suppressed on all addressing and protection exceptions.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Addressing (page-table entry)
- Privileged operation
- Protection (fetch and store, page-table entry, key-controlled protection, and low-address protection)

**Programming Notes:**

1. The selective clearing of entries may be implemented in different ways, depending on the model, and, in general, more entries may be cleared than the minimum number required. Some models may clear all entries which contain the page-frame real address obtained from the page-table entry in storage. Others may clear all entries which contain the designated page index, and some implementations may clear precisely the minimum number of entries required. Therefore, in order for a program to operate on all models, the program should not take advantage of any properties obtained by a less selective clearing on a particular model.

2. The clearing of TLB entries may make use of the page-frame real address in the page-table entry. Therefore, if the page-table entry, when in the attached state, ever contained a page-frame real address that is different from the current value, copies of entries containing the previous values may remain in the TLB.

3. INVALIDATE PAGE TABLE ENTRY cannot be safely used to update a shared location in main storage if the possibility exists that another CPU or a channel program may also be updating the location.
LOAD ADDRESS SPACE
PARAMETERS

LASP  D₁(B₁), D₂(B₂)  [SSE]

<table>
<thead>
<tr>
<th>'E500'</th>
<th>B₁</th>
<th>D₁</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The first operand contains values to be loaded into control registers 3 and 4, including a secondary ASN (SASN) and a primary ASN (PASN), and, possibly, a secondary ASTE instance number (SASTEIN) and a primary ASTE instance number (PASTEIN). Execution of the instruction consists in performing four major steps: PASN translation, SASN translation, SASN authorization, and control-register loading. Each of these steps may or may not be performed, depending on the outcome of certain tests and on the setting of bits 61-63 of the second-operand address. The first three of these steps, when performed and successful, obtain additional values, that are loaded into control registers 1, 5, and 7. When the first three steps are not successful when performed, no control registers are changed, and the reason is indicated in the condition code.

When the ASN-and-LX-reuse facility is not installed, or is installed but is not enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the first operand is a doubleword containing a PSW-key mask (PKM), a SASN, an authorization index (AX), and a PASN. When the ASN-and-LX-reuse facility is installed and enabled, the first operand is two consecutive doublewords containing those four values and also a SASTEIN and a PASTEIN.

The primary ASN may be translated by means of the ASN-translation tables to obtain a primary-ASTE (PASTE) origin (PASTEO) and, from the PASTE, a primary ASCE (PASCE). The secondary ASN may be translated by means of the ASN-translation tables to obtain a secondary ASCE (SASCE). If the first operand is two doublewords and PASN or SASN translation occurs, the ASTEIN in the located ASTE is checked for being equal to the PASTEIN or SASTEIN, respectively, in the first operand. An authority check for ensuring that the new AX is authorized to establish the new SASN can be prevented when SASN translation occurs, or it can be required and cause SASN translation even when SASN translation would not otherwise occur. The new AX, which is placed in a control register and is used in SASN authorization if that occurs, is the old AX if PASN translation has not occurred or is the AX in the located PASTE otherwise. However, in either of these cases, the new AX may be the AX specified in the first operand.

A doubleword first operand has the following format:

<table>
<thead>
<tr>
<th>PKM-d</th>
<th>SASN-d</th>
<th>AX-d</th>
<th>PASN-d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>32</td>
<td>48</td>
</tr>
</tbody>
</table>

A two-doubleword first operand has the following format:

<table>
<thead>
<tr>
<th>SASTEIN-d</th>
<th>PKM-d</th>
<th>SASN-d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PASTEIN-d</th>
<th>AX-d</th>
<th>PASN-d</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>96</td>
<td>112</td>
</tr>
</tbody>
</table>

The “d” stands for designated value and is used to distinguish these fields from other fields with similar names which are referred to in the definition. The current contents of the corresponding fields in the control registers are referred to as SASTEIN-old, PKM-old, SASN-old, etc. The updated contents of the control registers are referred to as SASTEIN-new, PKM-new, SASN-new, etc.

The second-operand address is not used to address data; instead, the rightmost three bits are used to control portions of the operation. The remainder of the second-operand address is ignored. Bits 61-63 of the second-operand address are used as follows:
<table>
<thead>
<tr>
<th>Bit</th>
<th>Function Specified in Second-Operand Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>ASN translation performed only when new ASN and old ASN are different.</td>
</tr>
<tr>
<td>62</td>
<td>AX associated with new PASN used.</td>
</tr>
<tr>
<td>63</td>
<td>SASN authorization performed.</td>
</tr>
</tbody>
</table>

* SASN translation and SASN authorization are performed only when SASN-d is not equal to PASN-d. When SASN-d is equal to PASN-d, the SASCE is set equal to the PASCE, the SASTEIN is set equal to the PASTEIN (if ASN-and-LX reuse is enabled), and no authorization is performed.

The operation of LOAD ADDRESS SPACE PARAMETERS is depicted in Figure 10-17 on page 10-44.

**Note:** In the following sections, the actions involving the PASTEIN and SASTEIN occur only if the ASN-and-LX-reuse facility is installed and is enabled by the ASN-and-LX-reuse control in control register 0.

**PASN Translation and Related Processing**

In the PASN-translation process, the PASN-d is translated by means of the ASN first table and the ASN second table. The PASTE0 resulting from PASN translation replaces the PASTE0 in control register 5. The ASCE in the located ASTE replaces the PASCE in control register 1. When bit 62 of the second-operand address is zero, the AX in the ASTE replaces the AX in control register 4. When bit 62 is one, AX-d replaces the AX in the control register. When ASN-and-LX reuse is enabled, PASTEIN-d is compared to the ASTEIN in the ASTE, and, when equal, replaces the PASTEIN in control register 4.

When bit 61 of the second-operand address is one, PASN translation is always performed. When bit 61 is zero, PASN translation is performed only if PASN-d is not equal to PASN-old. When bit 61 is zero and PASN-d is equal to PASN-old, the PASCE-old PASTE0-old, and PASTEIN-old are left unchanged in the control registers and become the PASCE-new and PASTE0-new, and PASTEIN-new, respectively.

In this case, if bit 62 is zero, then the AX-old is left unchanged in the control register and becomes the AX-new, or, if bit 62 is one, AX-new is set equal to AX-d.

The PASN translation follows the normal rules for ASN translation, except that the invalid bits, bit 0 in the ASN-first-table entry and bit 0 in the ASTE, when ones, do not result in an ASN-translation exception. When either of the invalid bits is one, condition code 1 is set. Condition code 1 is also set if PASN translation occurs, the ASN-and-LX-reuse facility is enabled, and PASTEIN-d is not equal to the ASTEIN in the ASTE. When a reason for setting condition code 1 does not exist and either the current primary space-switch-event-control bit in control register 1 is one or the space-switch-event-control bit in the ASTE is one, a space-switch event does not occur; instead, condition code 3 is set. When condition code 1 or 3 is set, the control registers remain unchanged.

The contents of the AX, ASCE, and ASTEIN fields in the ASTE which is accessed as a result of the PASN translation are referred to as AX-p, ASCE-p, and ASTEIN-p, respectively. The origin of the ASTE is referred to as PASTE0-p.

The description in this paragraph applies to use of the subspace-group facility. After ASCE-p has been obtained, if (1) the subspace-group-control bit, bit 54 in ASCE-p, is one, (2) the dispatchable unit is subspace active, and (3) PASTE0-p designates the ASTE for the base space of the dispatchable unit, then a copy of ASCE-p, called ASCE-rp, is made, and bits 0-55 and 58-63 of ASCE-rp are replaced by the same bits of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. Further details are in "Subspace-Replacement Operations" on page 5-68. If bit 0 in the subspace ASTE is one, or if the ASTE sequence number (ASTESN) in the subspace ASTE does not equal the subspace ASTESN in the dispatchable-unit control table, an exception is not recognized; instead, condition code 1 is set, and the control registers remain unchanged.

**SASN Translation and Related Processing**

In the SASN-translation process, the SASN-d is translated by means of the ASN first table and the ASN second table. The ASCE field obtained from
the ASTE subsequently replaces the secondary
ASCE (SASCE) in control register 7. When
ASN-and-LX reuse is enabled, SASTEIN-d is com-
pared to the ASTEIN in the ASTE, and, when
equal, replaces the SASTEIN in control register 3.

SASN translation is performed only when, but not necessarily when, SASN-d is not equal to
PASN-d. When SASN-d is equal to PASN-d,
SASCE-new and SASTEIN-new are set equal to
PASCE-new and PASTEIN-new, respectively. In
this case, there is not a test of whether
SASTEIN-d is equal to PASTEIN-d; SASTEIN-d is
ignored. When SASN-d is not equal to PASN-d
and is equal to SASN-old, bit 61 (force ASN trans-
lation) is zero, and bit 63 (skip ASN authori-
zation) is one, SASN translation is not performed,
and SASCE-old and SASTEIN-old become
SASCE-new and SASTEIN-new, respectively. In
this case, there is not a test of whether
SASTEIN-d is equal to SASTEIN-old; SASTEIN-d
is ignored.

SASN translation is performed in each of the fol-
lowing cases:

- SASN-d is not equal to PASN-d or SASN-old.
- SASN-d is not equal to PASN-d but is equal
to SASN-old, and either bit 61 (force ASN trans-
lation) of the second-operand address is one or bit 63 (skip secondary authority test) of
that address is zero. (The translation must be
performed when bit 63 is zero in order to
obtain the ATO and ATL from the SASTE.)

The SASN translation follows the normal rules for
ASN translation, except that the invalid bits, bit 0
in the ASN-first-table entry and bit 0 in the ASTE,
when ones, do not result in an ASN-translation
exception. When either of the invalid bits is one,
condition code 2 is Condition code 2 is also set if
SASN translation occurs, the ASN-and-LX-reuse
facility is enabled, and SASTEIN-d is not equal to
the ASTEIN in the ASTE. When condition code 2
is set, the control registers remain unchanged.

The contents of the ASCE, ATO, ATL and
ASTEIN fields in the ASTE which is accessed as
a result of the SASN translation are referred to as
ASCE-s, ATO-s, ATL-s, and ASTEIN-s, respect-
ively. The origin of the ASTE is referred to as
SASTEO-s.

The description in this paragraph applies to use of
the subspace-group facility. After ASCE-s has
been obtained, if (1) the subspace-group-control
bit, bit 54 in ASCE-s, is one, (2) the dispatchable
unit is subspace active, and (3) SASTEO-s design-
ates the ASTE for the base space of the
dispatchable unit, then a copy of ASCE-s, called
ASCE-rs, is made, and bits 0-55 and 58-63 of
ASCE-rs are replaced by the same bits of the
ASCE in the ASTE for the subspace in which the
dispatchable unit last had control. Further details
are in “Subspace-Replacement Operations” on
page 5-68. If bit 0 in the subspace ASTE is one,
or if the ASTE sequence number (ASTESN) in the
subspace ASTE does not equal the subspace
ASTEIN in the dispatchable-unit control table, an
exception is not recognized; instead, condition
code 2 is set, and the control registers remain
unchanged.

SASN Authorization

SASN authorization is performed when bit 63 of
the second-operand address is zero and SASN-d
is not equal to PASN-d; it is performed in this
case regardless of whether SASN-d is equal to
SASN-old. When SASN-d is equal to PASN-d or
when bit 63 of the second-operand address is one, SASN authorization is not performed.

SASN authorization is performed by using ATO-s,
ATL-s, and the intended value for AX-new. When
bit 62 of the second-operand address is zero and
PASN translation was performed, the intended
value for AX-new is AX-p. When bit 62 of that
address is zero and PASN translation was not
performed, the AX is not changed, and AX-new is
the same as AX-old. When bit 62 of that address
is one, the intended value for AX-new is AX-d.
SASN authorization follows the rules for sec-
ondary authorization as described in
“ASN-Authorization Process” on page 3-31. If the
SASN is not authorized (that is, the authority-table
length is exceeded, or the selected bit is zero),
condition code 2 is set, and none of the control
registers is updated.

Control-Register Loading

When the PASN-translation and SASN-translation
functions and related functions, and the
SASN-authorization functions and subspace-
replacement operations, if called for in the instruc-
tion execution, are performed without encountering
any exceptions or exception conditions, the execution is completed by replacing the contents of control registers 1, 3, 4, 5, and 7 with the new values, and condition code 0 is set. The control registers are loaded as follows.

The PSW-key-mask, bits 32-47, and SASN, bits 48-63, in control register 3 are replaced by the contents of the PKM-d and SASN-d fields of the first operand.

If ASN-and-LX reuse is enabled, the SASTEIN, bits 0-31, in control register 3 is replaced as follows:

- When SASN translation is performed, by SASTEIN-d.
- When SASN translation is not performed because SASN-d is equal to PASN-d, by PASTEIN-new.
- When SASN translation is not performed because (1) SASN-d is not equal to PASN-d but is equal to SASN-old, (2) bit 61 (force ASN translation) of the second-operand address is zero, and (3) bit 63 (skip secondary authority test) of the second-operand address is one, the SASTEIN remains unchanged.
- When ASN-and-LX reuse is not enabled, bits 0-31 of control register 3 always remain unchanged.

The PASN, bits 48-63 of control register 4, is replaced by the PASN-d of the first operand. If ASN-and-LX reuse is enabled, the PASTEIN, bits 0-31 of control register 4, is replaced by the PASTEIN-d of the first operand; otherwise, bits 0-31 of the register remain unchanged.

The authorization index, bits 32-47 of control register 4, is replaced as follows:

- When bit 62 of the second-operand address is one, by AX-d.
- When bit 62 of the second-operand address is zero and PASN translation is performed, by AX-p.
- When bit 62 of the second-operand address is zero and PASN translation is not performed, the authorization index remains unchanged.

The primary address-space-control element (PASCE) in control register 1 and the primary-ASN-second-table-entry origin (PASTEO) in control register 5 are replaced as follows:

- When PASN translation is performed, the PASCE in control register 1 is replaced by the ASCE-p obtained as a result of PASN translation, except that it is replaced by ASCE-rp if a subspace-replacement operation was performed on ASCE-p. Also, the PASTEO in control register 5 is replaced by PASTEO-p.

The PASTEO-p is placed in bit positions 33-57 of control register 5, and zeros are placed in bit positions 32 and 58-63. Bits 0-31 of the register remain unchanged.

- When PASN translation is not performed, the contents of control registers 1 and 5 remain unchanged.

The secondary address-space-control element (SASCE) in control register 7 is replaced as follows:

- When SASN-d equals PASN-d, by the new contents of control register 1, the PASCE. The new contents may be PASCE-old, ASCE-p, or ASCE-rp.
- When SASN translation is performed, by ASCE-s, or by ASCE-rs if a subspace-replacement operation was performed on ASCE-s.

When SASN-d does not equal PASN-d and SASN translation is not performed, the SASCE remains unchanged.

Other Condition-Code Settings

When PASN translation is called for and cannot be completed because bit 0 is one in either the ASN-first-table entry or the ASTE, or if it can be completed but (1) ASN-and-LX reuse is enabled and PASTEIN-d does not equal the ASTEIN in the ASTE or (2) a subspace-replacement-exception condition exists due to bit 0 or the ASTE sequence number in the subspace ASTE during a subspace-replacement operation on the ASCE-p, condition code 1 is set, and the control registers are not changed.

When PASN translation is called for and completed and any required PASTEIN-d comparison and subspace-replacement operations on the ASCE-p are also completed, and then either (1) the current primary space-switch-event-control...
bit, bit 57 of control register 1, is one or (2) the space-switch-event-control bit in the ASTE designated by PASTEO-p is one, condition code 3 is set, and the control registers are not changed.

When SASN translation is called for and the translation cannot be completed because bit 0 is one in either the ASN-first-table entry or the ASTE, or if it can be completed but (1) ASN-and-LX reuse is enabled and SASTEIN-d does not equal the ASTEIN in the ASTE, (2) a subspace-replacement-exception condition exists due to bit 0 or the ASTE sequence number in the subspace ASTE during a subspace-replacement operation on the ASCE-s, or (3) SASN authorization is called for and the SASN is not authorized, or condition code 2 is set, and the control registers are not changed.

**Special Conditions**

The instruction can be executed only when the ASN-translation control, bit 44 of control register 14, is one. If the ASN-translation-control bit is zero, a special-operation exception is recognized.

The first operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

The operation is suppressed on all addressing and protection exceptions.

**Program Exceptions:**

- Access (fetch, operand 1)
- Addressing (ASN-first-table entry, ASN-second-table entry, authority-table entry, dispatchable-unit control table)
- Privileged operation
- Special operation
- Specification

**Programming Notes:**

1. Bits 61 and 63 in the second-operand address are intended primarily to provide improved performance for those cases where the associated action is unnecessary.

When bit 61 is set to zero, the action of the instruction is based on the assumption that the current values for PASCE-old, PASTEO-old, and AX-old are consistent with PASN-old and that SASCE-old is consistent with SASN-old. When this is not the case, bit 61 should be set to one.

Bit 63, when one, eliminates the SASN-authorization test. The program may be able to determine in certain cases that the SASN is authorized, either because of prior use or because the AX being loaded is authorized to access all address spaces.

2. The SASN-translation and SASN-authorization steps are not performed when SASN-d is equal to PASN-d. This is consistent with the action in SET SECONDARY ASN to current primary (SSAR-cp), which does not perform the translation or ASN authorization.

3. The storage-operand references for LOAD ADDRESS SPACE PARAMETERS may be multiple-access references. (See “Storage Operand Consistency” on page 5-98)

4. See Figure 10-15 on page 10-42 for a listing of abbreviations used in this instruction description.
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.

7.A Access exceptions for second and third instruction halfwords.

7.B.1 Privileged-operation exception.

7.B.2 Special-operation exception due to the ASN-translation control, bit 34 of control register 14, being zero.

8. Specification exception.


10. PASN translation and related processing (when performed).

10.1 Addressing exception for access to ASN-first-table entry.

10.2 Condition code 1 due to I bit (bit 0) in ASN-first-table entry being one.

10.3 Addressing exception for access to ASN-second-table entry.

10.4 Condition code 1 due to (1) I bit (bit 0) in ASN-second-table entry (ASTE) being one or (2) ASN-and-LX reuse enabled and primary ASTE instance number (PASTEIN) in first operand not being equal to ASTEIN in ASTE.

10.5 Addressing exception for access to dispatchable-unit control table.

10.6 Addressing exception for access to subspace ASN-second-table entry.

10.7 Condition code 1 due to I bit (bit 0) in subspace ASN-second-table entry being one.

10.8 Condition code 1 due to subspace ASN-second-table-entry sequence number (SSASTESN) in dispatchable-unit control table not being equal to ASTESN in subspace ASN-second-table entry.

10.9 Condition code 3 due to either the old or new space-switch-event-control bit being one.

11. SASN translation and related processing (when performed).

11.1 Addressing exception for access to ASN-first-table entry.

11.2 Condition code 2 due to I bit (bit 0) in ASN-first-table entry being one.

11.3 Addressing exception for access to ASN-second-table entry.

11.4 Condition code 2 due to (1) I bit (bit 0) in ASN-second-table entry (ASTE) being one or (2) ASN-and-LX reuse enabled and secondary ASTE instance number (SASTEIN) in first operand not being equal to ASTEIN in ASTE.

12.A Execution of secondary authorization (when performed).

12.A.1 Condition code 2 due to authority-table entry being outside table.

12.A.2 Addressing exception for access to authority-table entry.

Figure 10-14 (Part 1 of 2). Priority of Execution: LOAD ADDRESS SPACE PARAMETERS
12.A.3 Condition code 2 due to $S$ bit in authority-table entry being zero.

12.B.1 Addressing exception for access to dispatchable-unit control table.

12.B.2 Addressing exception for access to subspace ASN-second-table entry.

12.B.3 Condition code 2 due to $I$ bit (bit 0) in subspace ASN-second-table entry being one.

12.B.4 Condition code 2 due to subspace ASN-second-table-entry sequence number (SSASTESN) in dispatchable-unit control table not being equal to ASTESN in subspace ASN-second-table entry.

Figure 10-14 (Part 2 of 2). Priority of Execution: LOAD ADDRESS SPACE PARAMETERS

<table>
<thead>
<tr>
<th>First-Operand Bit Positions when ASN-and-LX Reuse Not Enabled</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15</td>
<td>PKM-d</td>
</tr>
<tr>
<td>16-31</td>
<td>SASN-d</td>
</tr>
<tr>
<td>32-47</td>
<td>AX-d</td>
</tr>
<tr>
<td>48-63</td>
<td>PASN-d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First-Operand Bit Positions when ASN-and-LX Reuse Enabled</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>SASTEIN-d</td>
</tr>
<tr>
<td>32-47</td>
<td>PKM-d</td>
</tr>
<tr>
<td>48-63</td>
<td>SASN-d</td>
</tr>
<tr>
<td>64-95</td>
<td>PASTEIN-d</td>
</tr>
<tr>
<td>96-111</td>
<td>AX-d</td>
</tr>
<tr>
<td>112-127</td>
<td>PASN-d</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control-Register Number.Bit</th>
<th>Abbreviation for</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Previous Contents</td>
</tr>
<tr>
<td>1.0-63</td>
<td>PASCE-old</td>
</tr>
<tr>
<td>3.0-31</td>
<td>SASTEIN-old</td>
</tr>
<tr>
<td>3.32-47</td>
<td>PKM-old</td>
</tr>
<tr>
<td>3.48-63</td>
<td>SASN-old</td>
</tr>
<tr>
<td>4.0-31</td>
<td>PASTEIN-old</td>
</tr>
<tr>
<td>4.32-47</td>
<td>AX-old</td>
</tr>
<tr>
<td>4.48-63</td>
<td>PASN-old</td>
</tr>
<tr>
<td>5.33-57</td>
<td>PASTE0-old</td>
</tr>
<tr>
<td>7.0-63</td>
<td>SASCE-old</td>
</tr>
</tbody>
</table>

Explanation:
- The field is not used in this case.

1 ASCE-rp is formed from ASCE-p, and ASCE-rs is formed from ASCE-s, by a subspace-replacement operation.

Figure 10-15 (Part 2 of 2). Summary of Abbreviations for LOAD ADDRESS SPACE PARAMETERS
<table>
<thead>
<tr>
<th>PASN-d</th>
<th>Second-Operand-Address Bits</th>
<th>PASN Translation Performed</th>
<th>Result Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>PASCE-new</td>
</tr>
<tr>
<td>Equals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PASN-old</td>
<td>61</td>
<td>62</td>
<td>No</td>
</tr>
<tr>
<td>Yes 0 0</td>
<td>No</td>
<td>No</td>
<td>PASCE-old</td>
</tr>
<tr>
<td>Yes 0 1</td>
<td>No</td>
<td>No</td>
<td>PASCE-old</td>
</tr>
<tr>
<td>Yes 1 0</td>
<td>Yes</td>
<td>ASCE-p^2</td>
<td>AX-p</td>
</tr>
<tr>
<td>Yes 1 1</td>
<td>Yes</td>
<td>ASCE-p^2</td>
<td>AX-d</td>
</tr>
<tr>
<td>No - 0</td>
<td>Yes</td>
<td>ASCE-p^2</td>
<td>AX-p</td>
</tr>
<tr>
<td>No - 1</td>
<td>Yes</td>
<td>ASCE-p^2</td>
<td>AX-d</td>
</tr>
</tbody>
</table>

**Explanation:**

- Action in this case is the same regardless of the outcome of this comparison or of the setting of this bit.

1 Second-operand-address bits:
   - 61 Force ASN translation.
   - 62 Use AX from first operand.
   - 63 Skip secondary authority test.

2 PASCE-new is ASCE-rp (a copy of ASCE-p except with bits 0-55 and 58-63 replaced from the ASCE in the subspace ASTE), if subspace replacement is performed.

3 SASN authorization is performed using ATO-s, ATL-s, and AX-new.

4 SASCE-new is ASCE-rs (a copy of ASCE-s except with bits 0-55 and 58-63 replaced from the ASCE in the subspace ASTE), if subspace replacement is performed.
Figure 10-17. Execution of LOAD ADDRESS SPACE PARAMETERS

*: PASCE-tmp is ASCE-rp if subspace replacement occurred.
**: SASCE-tmp is ASCE-rs if subspace replacement occurred.

Note: Actions involving PASTEIN and SASTEIN occur only if ASN-and-LX reuse is enabled.
LOAD CONTROL

LCTL  R1, R3, D2 (B2)  [RS]

<table>
<thead>
<tr>
<th>'B7'</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Bit positions of the set of control registers starting with control register R1 and ending with control register R3 are loaded from storage beginning at the location designated by the second-operand address and continuing through as many locations as needed.

For LOAD CONTROL (LCTL), bit positions 32-63 of the control registers are loaded from successive words beginning at the second-operand address, and bits 0-31 of the registers remain unchanged.

For LOAD CONTROL (LCTLG), bit positions 0-63 of the control registers are loaded from successive doublewords beginning at the second-operand address. The control registers are loaded in ascending order of their register numbers, starting with control register R1 and continuing up to and including control register R3, with control register 0 following control register 15.

The information loaded into the control registers becomes active when instruction execution has ended.

The displacement for LCTL is treated as a 12-bit unsigned binary integer. The displacement for LCTLG is treated as a 20-bit signed binary integer.

Special Conditions

The second operand must be designated on a word boundary for LCTL or on a doubleword boundary for LCTLG; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2)
- Privileged operation
- Specification

Programming Notes:

1. To ensure that existing programs operate correctly if and when new facilities using additional control-register positions are defined, only zeros should be loaded in unassigned control-register positions.

2. Loading of control registers on some models may require a significant amount of time. This is particularly true for changes in significant parameters.

For example, the TLB may be cleared of entries as a result of changing or enabling the program-event-recording parameters in control registers 9-11. Where possible, the program should avoid unnecessary loading of control registers. In loading control registers 9-11, most models attempt to optimize for the case when the bits of control register 9 are zeros.

LOAD PSW

LPSW  D2 (B2)  [S]

<table>
<thead>
<tr>
<th>'B2'</th>
<th>//////////////</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The current PSW is replaced by a 16-byte PSW formed from the contents of the doubleword at the location designated by the second-operand address.

Bit 12 of the doubleword must be one; otherwise, a specification exception may be recognized, depending on the model.

Bits 0-32 of the doubleword, except with bit 12 inverted, are placed in bit positions 0-32 of the current PSW. Bits 33-63 of the doubleword are placed in bit positions 97-127 of the current PSW. Bits 33-96 of the current PSW are set to zeros.

A serialization and checkpoint-synchronization function is performed before or after the operand is fetched and again after the operation is completed.
Special Conditions

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized. A specification exception may be recognized if bit 12 of the operand is zero, depending on the model.

The PSW fields which are to be loaded by the instruction are not checked for validity before they are loaded, except for the optional checking of bit 12. However, immediately after loading, a specification exception is recognized, and a program interruption occurs, when any of the following is true for the newly loaded PSW:

- Any of bits 0, 2-4, 24-30 is a one.
- Bits 31 and 32 are both zero, and bits 97-103 are not all zeros.
- Bits 31 and 32 are one and zero, respectively.

In these cases, the operation is completed, and the resulting instruction-length code is 0.

The test for a specification exception after the PSW is loaded is described in "Early Exception Recognition" on page 6-9. It may be considered as occurring early in the process of preparing to execute the subsequent instruction.

The operation is suppressed on all addressing and protection exceptions.

**Resulting Condition Code:** The code is set as specified in the new PSW loaded.

**Program Exceptions:**

- Access (fetch, operand 2)
- Privileged operation
- Specification

**Programming Note:** The second operand should have the format of an ESA/390 PSW. A specification exception will be recognized during or after the execution of LOAD PSW if bit 12 of the operand is zero.

LOAD PSW EXTENDED

```
LPSW   D2(B2) [S]

┌────────────────┬────┬────────────┐
│ 'B2B2' │ B2 │ D2    │
└────────────────┴────┴────────────┘
```

The current PSW is replaced by the contents of the 16-byte second operand.

A serialization and checkpoint-synchronization function is performed before or after the operand is fetched and again after the operation is completed.

Special Conditions

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

The value which is to be loaded by the instruction is not checked for validity before it is loaded. However, immediately after loading, a specification exception is recognized, and a program interruption occurs, when any of the following is true for the newly loaded PSW:

- Any of the unassigned bits (0, 2-4, 24-30, or 33-63) is a one.
- Bit 12 is a one.
- Bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros.
- Bits 31 and 32 are both zero, and bits 64-103 are not all zeros.
- Bits 31 and 32 are one and zero, respectively.

In these cases, the operation is completed, and the resulting instruction-length code is zero.

The test for a specification exception after the PSW is loaded is described in "Early Exception Recognition" on page 6-9. It may be considered as occurring early in the process of preparing to execute the subsequent instruction.

The operation is suppressed on all addressing and protection exceptions.

**Resulting Condition Code:** The code is set as specified in the new PSW loaded.
Program Exceptions:

- Access (fetch, operand 2)
- Privileged operation
- Specification

LOAD REAL ADDRESS

LRA  \( R_1, D_2(X_2, B_2) \)  [RX]

<table>
<thead>
<tr>
<th>'B1'</th>
<th>( R_1 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( D_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

LRAY  \( R_1, D_2(X_2, B_2) \)  [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>( R_1 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( DL_2 )</th>
<th>( DH_2 )</th>
<th>'13'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>47</td>
</tr>
</tbody>
</table>

LRAG  \( R_1, D_2(X_2, B_2) \)  [RXY]

<table>
<thead>
<tr>
<th>'E3'</th>
<th>( R_1 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( DL_2 )</th>
<th>( DH_2 )</th>
<th>'03'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>47</td>
</tr>
</tbody>
</table>

For LOAD REAL ADDRESS (LRA, LRAY) in the 24-bit or 31-bit addressing mode, if bits 0-32 of the 64-bit real address corresponding to the second-operand virtual address are all zeros, bits 32-63 of the real address are placed in bit positions 32-63 of general register \( R_1 \), and bits 0-31 of the register remain unchanged. If bits 0-32 of the real address are not all zeros, a special-operation exception is recognized.

For LRA or LRAY in the 64-bit addressing mode, and for LOAD REAL ADDRESS (LRAG) in any addressing mode, the 64-bit real address corresponding to the second-operand virtual address is placed in general register \( R_1 \).

The virtual address specified by the \( X_2 \), \( B_2 \), and \( D_2 \) fields is translated by means of the dynamic-address-translation facility, regardless of whether DAT is on or off.

The displacement for LRA is treated as a 12-bit unsigned binary integer. The displacement for LRAY and LRAG is treated as a 20-bit signed binary integer.

DAT is performed by using an address-space-control element that depends on the current value of the address-space-control bits, bits 16 and 17 of the PSW, as shown in the following table:

### PSW

<table>
<thead>
<tr>
<th>Bits 16</th>
<th>Address-Space-Control Element Used by DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Contents of control register 1</td>
</tr>
<tr>
<td>10</td>
<td>Contents of control register 7</td>
</tr>
<tr>
<td>01</td>
<td>The address-space-control element obtained by applying the access-register-translation (ART) process to the access register designated by the ( B_2 ) field</td>
</tr>
<tr>
<td>11</td>
<td>Contents of control register 13</td>
</tr>
</tbody>
</table>

ART and DAT may be performed with the use of the ART-lookaside buffer (ALB) and translation-lookaside buffer (TLB), respectively.

The virtual-address computation is performed according to the current addressing mode, specified by bits 31 and 32 of the current PSW.

The addresses of the region-table entry or entries, if used, and of the segment-table entry and page-table entry are treated as 64-bit addresses regardless of the current addressing mode. It is unpredictable whether the addresses of these entries are treated as real or absolute addresses.

Condition code 0 is set when both ART, if applicable, and DAT can be completed and a special-operation exception is not recognized, that is, when an address-space-control element can be obtained, the entry in each DAT table lies within the table and has a zero I bit, and, for LRA or LRAY in the 24-bit or 31-bit addressing mode, bits 0-32 of the resulting real address are zeros. The translated address is not inspected for boundary alignment or for addressing or protection exceptions.

When PSW bits 16 and 17 are 01 binary and an address-space-control element cannot be obtained because of a condition that would normally cause one of the exceptions shown in the following table, (1) the interruption code assigned to the exception is placed in bit positions 48-63 of general register \( R_1 \), bit 32 of this register is set to one, bits 33-47 are set to zeros, and bits 0-31 remain unchanged, and (2) the instruction is completed by setting condition code 3.
When ART is completed normally, the operation is continued through the performance of DAT.

When the segment-table entry is outside the table and bits 0-32 of the real or absolute address of the entry are all zeros, condition code 3 is set, bits 32-63 of the entry address are placed in bit positions 32-63 of general register R₁, and bits 0-31 of the register remain unchanged. If bits 0-32 of the address are not all zeros, the result is as shown in the next table below.

For LRA or LRAY in the 64-bit addressing mode or LRAG in any addressing mode, when the I bit in the segment-table entry is one, condition code 1 is set, and the 64-bit real or absolute address of the segment-table entry is placed in general register R₁. In this case except that LRA or LRAY is in the 24-bit or 31-bit addressing mode, if bits 0-32 of the address of the segment-table entry are all zeros, the result is the same except that bits 0-31 of general register R₁ remain unchanged. If bits 0-32 of the address are not all zeros, the result is as shown in the next table below.

For LRA or LRAY in the 64-bit addressing mode or LRAG in any addressing mode, when the I bit in the page-table entry is one, condition code 2 is set, and the 64-bit real or absolute address of the page-table entry is placed in general register R₁. In this case except that LRA or LRAY is in the 24-bit or 31-bit addressing mode, if bits 0-32 of the address of the page-table entry are all zeros, the result is the same except that bits 0-31 of general register R₁ remain unchanged. If bits 0-32 of the address are not all zeros, the result is as shown in the next table below.

A segment-table-entry or page-table-entry address placed in general register R₁ is real or absolute in accordance with the type of address that was used during the attempted translation.

If a condition exists that would normally cause one of the exceptions shown in the following table, (1) the interruption code assigned to the exception is placed in bit positions 48-63 of general register R₁, bit 32 of this register is set to one, bits 33-47 are set to zeros, and bits 0-31 remain unchanged, and (2) the instruction is completed by setting condition code 3.

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Cause</th>
<th>Code (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALET specification</td>
<td>Access-list-entry-token (ALET) bits 0-6 not all zeros</td>
<td>0028</td>
</tr>
<tr>
<td>ALEN translation</td>
<td>Access-list entry (ALE) outside list or invalid (bit 0 is one)</td>
<td>0029</td>
</tr>
<tr>
<td>ALE sequence</td>
<td>ALE sequence number (ALESN) in ALET not equal to ALESN in ALE</td>
<td>002A</td>
</tr>
<tr>
<td>ASTE validity</td>
<td>ASN-second-table entry (ASTE) invalid (bit 0 is one)</td>
<td>002B</td>
</tr>
<tr>
<td>ASTE sequence</td>
<td>ASTE sequence number (ASTESN) in ALE not equal to ASTESN in ASTE</td>
<td>002C</td>
</tr>
<tr>
<td>Extended authority</td>
<td>ALE private bit not zero, ALE authorization index (ALEAX) not equal to extended authorization index (EAX), and secondary bit selected by EAX either outside authority table or zero</td>
<td>002D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Cause</th>
<th>Code (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALET specification</td>
<td>Access-list-entry-token (ALET) bits 0-6 not all zeros</td>
<td>0028</td>
</tr>
<tr>
<td>ALEN translation</td>
<td>Access-list entry (ALE) outside list or invalid (bit 0 is one)</td>
<td>0029</td>
</tr>
<tr>
<td>ALE sequence</td>
<td>ALE sequence number (ALESN) in ALET not equal to ALESN in ALE</td>
<td>002A</td>
</tr>
<tr>
<td>ASTE validity</td>
<td>ASN-second-table entry (ASTE) invalid (bit 0 is one)</td>
<td>002B</td>
</tr>
<tr>
<td>ASTE sequence</td>
<td>ASTE sequence number (ASTESN) in ALE not equal to ASTESN in ASTE</td>
<td>002C</td>
</tr>
<tr>
<td>Extended authority</td>
<td>ALE private bit not zero, ALE authorization index (ALEAX) not equal to extended authorization index (EAX), and secondary bit selected by EAX either outside authority table or zero</td>
<td>002D</td>
</tr>
</tbody>
</table>
A special-operation exception is recognized when, for LRA or LRAY in the 24-bit or 31-bit addressing mode, bits 0-32 of the resultant 64-bit real address are not all zeros.

An addressing exception is recognized when the address used by ART to fetch the effective access-list designation or the ALE, ASTE, or authority-table entry designates a location which is not available in the configuration or when the address used to fetch the region-table entry or entries, if any, segment-table entry, or page-table entry designates a location which is not available in the configuration.

A translation-specification exception is recognized when an accessed region-table entry or the segment-table entry or page-table entry has a zero I bit and a format error, that is, when any of the reasons listed in “Translation-Specification Exception” on page 6-36 applies.

A carry out of bit position 0 as a result of the addition done to compute the address of a region-table entry or the segment-table entry may be ignored or may result in an addressing exception.

The operation is suppressed on all addressing exceptions.

**Resulting Condition Code:**

0  Translation available
1  Segment-table entry invalid (I bit one)
2  Page-table entry invalid (I bit one)
3  Address-space-control element not available, region-table entry outside table or invalid (I bit one), segment-table entry outside table, or, for LRA and LRAY only, and only in 24-bit or 31-bit addressing mode when bits 0-32 of entry address not all zeros), segment-table entry or page-table entry invalid (I bit one)

**Program Exceptions:**

- Addressing (effective access-list designation, access-list entry, ASN-second-table entry, authority-table entry, region-table entry, segment-table entry, or page-table entry)
- Operation (LRAY, if the long-displacement facility is not installed)
- Privileged operation
- Special operation (LRA, LRAY only)
Translation specification

Programming Notes:

1. Caution must be exercised in the use of LOAD REAL ADDRESS in a multiprocessing configuration. Since INVALIDATE PAGE TABLE ENTRY may set the I bit in storage to one before causing the corresponding entries in TLBs of other CPUs to be cleared, the simultaneous execution of LOAD REAL ADDRESS on this CPU and INVALIDATE PAGE TABLE ENTRY on another CPU may produce inconsistent results. Because LOAD REAL ADDRESS may access the tables in storage, the page-table entry may appear to be invalid (condition code 2) even though the corresponding TLB entry has not yet been cleared, and the TLB entry may remain in the TLB until the completion of INVALIDATE PAGE TABLE ENTRY on the other CPU. There is no guaranteed limit to the number of instructions which may be executed between the completion of LOAD REAL ADDRESS and the TLB being cleared of the entry.

2. Figure 10-18 on page 10-51 summarizes the resulting contents of general register R1 and the condition code.
<table>
<thead>
<tr>
<th>Exception/Cause/Entry-Address Size or Resultant-Real-Address Size</th>
<th>General Register R1 Contents and Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRA or LRAY in 24-Bit or 31-Bit Addressing Mode</td>
</tr>
<tr>
<td></td>
<td>0-31  32  33-47  48-63  CC</td>
</tr>
<tr>
<td>ALET specification</td>
<td>U  1  0s  002B  3</td>
</tr>
<tr>
<td>ALEN translation</td>
<td>U  1  0s  0029  3</td>
</tr>
<tr>
<td>ALE sequence</td>
<td>U  1  0s  002A  3</td>
</tr>
<tr>
<td>ASTE validity</td>
<td>U  1  0s  002B  3</td>
</tr>
<tr>
<td>ASTE sequence</td>
<td>U  1  0s  002C  3</td>
</tr>
<tr>
<td>Extended authority</td>
<td>U  1  0s  002D  3</td>
</tr>
<tr>
<td>ASCE type</td>
<td>U  1  0s  003B  3</td>
</tr>
<tr>
<td>Region first trans.</td>
<td>U  1  0s  0039  3</td>
</tr>
<tr>
<td>Region second trans.</td>
<td>U  1  0s  003A  3</td>
</tr>
<tr>
<td>Region third trans.</td>
<td>U  1  0s  003B  3</td>
</tr>
<tr>
<td>Segment translation/entry outside table/entry address &lt; 2GB</td>
<td>U  0  EA3  EA4  3</td>
</tr>
<tr>
<td>Segment translation/entry outside table/entry address &gt;= 2GB</td>
<td>U  1  0s  0010  3</td>
</tr>
<tr>
<td>Segment translation/I bit one/entry address &lt; 2GB</td>
<td>U  0  EA3  EA4  1</td>
</tr>
<tr>
<td>Segment translation/I bit one/entry address &gt;= 2GB</td>
<td>U  1  0s  0010  3</td>
</tr>
<tr>
<td>Page translation/I bit one/entry address &lt; 2GB</td>
<td>U  0  EA3  EA4  2</td>
</tr>
<tr>
<td>Page translation/I bit one/entry address &gt;= 2GB</td>
<td>U  1  0s  0011  3</td>
</tr>
<tr>
<td>Real Address &lt; 2GB</td>
<td>U  0  RA3  RA4  0</td>
</tr>
<tr>
<td>Real Address &gt;= 2GB</td>
<td>Special-Operation Exception</td>
</tr>
</tbody>
</table>

Figure 10-18 (Part 1 of 2). Summary of Results: LOAD REAL ADDRESS
**Explanation:**

EA1 Bits 0-31 of the entry address.
EA2 Bit 32 of the entry address.
EA3 Bits 33-47 of the entry address.
EA4 Bits 48-63 of the entry address.
RA1 Bits 0-31 of the resultant real address.
RA2 Bit 32 of the resultant real address.
RA3 Bits 33-47 of the resultant real address.
RA4 Bits 48-63 of the resultant real address.
U Unchanged.

**Figure 10-18 (Part 2 of 2). Summary of Results: LOAD REAL ADDRESS**

**LOAD USING REAL ADDRESS**

<table>
<thead>
<tr>
<th>LURA</th>
<th>R1, R2</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B24B'</td>
<td>//.....</td>
<td>R1 R2</td>
</tr>
<tr>
<td>0 16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LURAG</th>
<th>R1, R2</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B905'</td>
<td>//.....</td>
<td>R1 R2</td>
</tr>
<tr>
<td>0 16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For LOAD USING REAL ADDRESS (LURA), the word at the real-storage location addressed by the contents of general register R2 is placed in bit positions 32-63 of general register R1, and the contents of bit positions 0-31 remain unchanged. For LOAD USING REAL ADDRESS (LURAG), the doubleword at that real-storage location is placed in bit positions 0-63 of general register R1.

In the 24-bit addressing mode, bits 40-63 of general register R2 designate the real-storage location, and bits 0-39 of the register are ignored. In the 31-bit addressing mode, bits 33-63 of general register R2 designate the real-storage location, and bits 0-33 of the register are ignored. In the 64-bit addressing mode, bits 0-63 of general register R2 designate the real-storage location.

Because it is a real address, the address designating the storage word or doubleword is not subject to dynamic address translation.

**Special Conditions**

The contents of general register R2 must designate a location on a word boundary for LURA or on a doubleword boundary for LURAG; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Addressing (address specified by general register R2)
- Privileged operation
- Protection (fetch, operand 2, key-controlled protection)
- Specification

**MODIFY STACKED STATE**

<table>
<thead>
<tr>
<th>MSTA</th>
<th>R1</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B247'</td>
<td>//...</td>
<td>R1 ///</td>
</tr>
<tr>
<td>0 16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The contents of bit positions 32-63 of the pair of general registers designated by the R1 field are placed in the modifiable area, byte positions 152-159, of the last state entry in the linkage stack.

The R1 field designates the even-numbered register of an even-odd pair of general registers.

The last state entry is located as described in "Unstacking Process" on page 5-86. The state entry remains in the linkage stack, and the linkage-stack-entry address in control register 15 remains unchanged.
Key-controlled protection does not apply to the references to the linkage stack, but low-address and page protection do apply.

**Special Conditions**

A specification exception is recognized when \( R_1 \) is odd.

The CPU must be in the primary-space mode, access-register mode, or home-space mode; otherwise, a special-operation exception is recognized.

A stack-empty, stack-specification, or stack-type exception may be recognized during the unstacking process.

The operation is suppressed on all addressing and protection exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-19.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch and store, except for key-controlled protection, linkage-stack entry)
- Special operation
- Specification
- Stack empty
- Stack specification
- Stack type

---

| 1.-6. | Exceptions with the same priority as the priority of program-interruption conditions for the general case. |
| 7.B  | Special-operation exception due to DAT being off or the CPU being in secondary-space mode. |
| 8.A  | Specification exception due to \( R_1 \) being odd. |
| 8.B.1| Access exceptions (fetch) for entry descriptor of the current linkage-stack entry. |
| 8.B.2| Stack-type exception due to current entry not being a state entry or header entry. |

**Note:** Exceptions 8.B.3-8.B.7 can occur only if the current entry is a header entry.

| 8.B.3| Access exceptions (fetch) for second word of the header entry. |
| 8.B.4| Stack-empty exception due to backward stack-entry validity bit in the header entry being zero. |
| 8.B.5| Access exceptions (fetch) for entry descriptor of preceding entry, which is the entry designated by the backward stack-entry address in the current (header) entry. |
| 8.B.6| Stack-specification exception due to preceding entry being a header entry. |
| 8.B.7| Stack-type exception due to preceding entry not being a state entry. |
| 8.B.8| Access exceptions (store) for the modifiable area of the state entry. |

Figure 10-19. Priority of Execution: MODIFY STACKED STATE
MOVE PAGE

MVPG R1,R2 [RRE]

The first operand is replaced by the second operand. The first and second operands both are 4K bytes on 4K-byte boundaries. The results are indicated in the condition code. The accesses to the first-operand location or the second-operand location, but not to both locations, may be performed by using the key specified in general register 0; otherwise, the accesses to an operand location are performed by using the PSW key.

The location of the leftmost byte of the first operand and second operand is designated by the contents of general registers R/basesuperone and R/basesupertwo, respectively.

The handling of the addresses in general registers R1 and R2 depends on the addressing mode. In the 24-bit addressing mode, the contents of bit positions 40-51 of a general register, with 12 rightmost zeros appended, are the address, and bits 0-39 and 52-63 in the register are ignored. In the 31-bit addressing mode, the contents of bit positions 33-51 of a general register, with 12 rightmost zeros appended, are the address, and bits 0-32 and 52-63 in the register are ignored. In the 64-bit addressing mode, the contents of bit positions 0-51 of a general register, with 12 rightmost zeros appended, are the address, and bits 52-63 in the register are ignored.

Bits 56-59 of general register 0 are used as the specified access key. Bit 52 of general register 0, when one, specifies that the specified access key is to be used for accessing the first operand, and bit 53 specifies the same for the second operand. A specification exception is recognized if bits 52 and 53 are both ones. Bit 54 of general register 0 is a destination-reference-intention bit, and bit 55 is a condition-code-option bit. Bits 48-51 of general register 0 must be zeros; otherwise, a specification exception is recognized. Bits 0-47 and 60-63 of general register 0 are ignored.

The contents of the registers just described are shown in Figure 10-20 on page 10-55

When bit 52 of general register 0 is one, the fetch accesses to the second-operand location are performed by using the PSW key, and the store accesses to the first-operand location are performed by using the key specified in general register 0. When bit 53 of general register 0 is one, the fetch accesses to the second-operand location are performed by using the key specified in general register 0, and the store accesses to the first-operand location are performed by using the PSW key. When bits 52 and 53 are both zeros, the PSW key is used for accessing both operands.

When 4K bytes have been moved, condition code 0 is set.

When a page-translation-exception condition exists, the exception is not recognized if the condition-code-option bit, bit 55 in general register 0, is one; instead, condition code 1 or 2 is set. Condition code 1 is set if a page-translation-exception condition exists for the first operand and not for the second operand. Condition code 2 is set if a page-translation-exception condition exists for the second operand, regardless of whether the condition exists for the first operand.

When an access exception can be recognized for both operands, it is unpredictable for which operand an exception is recognized. If one of the exceptions is a page-translation exception that would cause condition code 1 or 2 to be set, it is unpredictable whether the access exception for the other operand is recognized or condition code 1 or 2 is set.

The references to main storage are not necessarily single-access references and are not necessarily performed in a left-to-right direction, as observed by other CPUs and by channel programs.

Special Conditions

In the problem state, when either bit 52 or bit 53 in general register 0 is one, the operation is performed only if the access key specified in general register 0 is valid, that is, if the corresponding PSW-key-mask bit in control register 3 is one. Otherwise, a privileged-operation exception is recognized. In the supervisor state, any value for the specified access key is valid. When bits 52...
and 53 are both zeros, the access key in general register 0 is not tested for validity.

In the problem state, when bits 52 and 53 in general register 0 are both ones and the access key in general register 0 is not permitted by the PSW-key mask, it is unpredictable whether a specification exception or a privileged-operation exception is recognized.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Data moved</td>
</tr>
<tr>
<td>1</td>
<td>Condition-code-option bit one, page-table entry for first operand invalid, and page-table entry for second operand valid</td>
</tr>
<tr>
<td>2</td>
<td>Condition-code-option bit one and page-table entry for second operand valid</td>
</tr>
<tr>
<td>3</td>
<td>Specification exception</td>
</tr>
</tbody>
</table>
Programming Notes:

1. MOVE PAGE, or a loop of MOVE PAGE instructions that moves multiple pages, may provide, on most models, better performance than a MOVE LONG instruction or a loop of MOVE (MVC) instructions that performs the same function. Whether MOVE PAGE provides better performance depends on control-program specifications and the method by which the control program handles page-translation exceptions.

2. The destination-reference-intention bit should be set to one when there is an intention to reference the first operand by means of an instruction other than MOVE PAGE. The bit may allow the control program to process a page-translation exception more efficiently.

3. The condition-code-option bit provides compatibility with the MOVE PAGE instruction of the ESA/390 move-page facility. The bit is for use by the MVS/ESA HSPSERV macro expansion.

4. The condition code set by the instruction normally need not be examined if the condition-code-option bit is zero.

5. See the definitions of real locations 162 and 168-175 under “Assigned Storage Locations” in Chapter 3, “Storage,” for a description of information stored during a program interruption due to a page-translation exception recognized by MOVE PAGE.

MOVE TO PRIMARY

```
MVCP D1(R1,B1),D2(B2),R3 [SS]
```

<table>
<thead>
<tr>
<th>'DA'</th>
<th>R1</th>
<th>R3</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

MOVE TO SECONDARY

```
MVCS D1(R1,B1),D2(B2),R3 [SS]
```

<table>
<thead>
<tr>
<th>'DB'</th>
<th>R1</th>
<th>R3</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The first operand is replaced by the second operand. One operand is in the primary address space, and the other is in the secondary address space. The accesses to the operand in the primary space are performed by using the PSW key; the accesses to the operand in the secondary space are performed by using the key specified by the third operand.

The addresses of the first and second operands are virtual, one operand address being translated by means of the primary address-space-control element and the other by means of the secondary address-space-control element. Operand-address translation is performed in the same way when the address-space-control bits in the current PSW specify either the primary-space mode or the secondary-space mode.

For MOVE TO PRIMARY, movement is to the primary space from the secondary space. The first-operand address is translated by using the primary address-space-control element, and the second-operand address is translated by using the secondary address-space-control element.

For MOVE TO SECONDARY, movement is to the secondary space from the primary space. The first-operand address is translated by using the secondary address-space-control element, and the second-operand address is translated by using the primary address-space-control element.

Bit positions 56-59 of general register R3 are used as the secondary-space access key. Bit positions 0-55 and 60-63 of the register are ignored.
General register \( R_1 \) contains an unsigned binary integer called the true length. In the 24-bit or 31-bit addressing mode, the true length is in bit positions 32-63 of the register, and the contents of bit positions 0-31 of the register are ignored. In the 64-bit addressing mode, the true length is in bit positions 0-63 of the register.

The contents of the general registers just described are as follows:

<table>
<thead>
<tr>
<th></th>
<th>24-Bit or 31-Bit Addressing Mode</th>
<th>64-Bit Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>True Length</td>
<td>////</td>
<td>True Length</td>
</tr>
<tr>
<td></td>
<td>0 32 63</td>
<td>0 63</td>
</tr>
<tr>
<td>( R_3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>///</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 56 60 63</td>
<td></td>
</tr>
</tbody>
</table>

The first and second operands are the same length, called the effective length. The effective length is equal to the true length or 256, whichever is less. Access exceptions for the first and second operands are recognized only for that portion of the operand within the effective length. When the effective length is zero, no access exceptions are recognized for the first and second operands, and no movement takes place.

Each storage operand is processed left to right. The storage-operand-consistency rules are the same as for MOVE (MVC), except that when the operands overlap in real storage, the use of the common real-storage locations is not necessarily recognized.

As part of the execution of the instruction, the value of the true length is used to set the condition code. If the true length is 256 or less, including zero, the true length and effective length are equal, and condition code 0 is set. If the true length is greater than 256, the effective length is 256, and condition code 3 is set.

For both MOVE TO PRIMARY and MOVE TO SECONDARY, a serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed.

### Special Conditions

Since the secondary space is accessed, the operation is performed only when the secondary-space control, bit 37 of control register 0, is one and DAT is on. When either the secondary-space control is zero or DAT is off, a special-operation exception is recognized. A special-operation exception is also recognized when the address-space-control bits in the current PSW specify the access-register or home-space mode.

In the problem state, the operation is performed only if the secondary-space access key is valid, that is, if the corresponding PSW-key-mask bit in control register 3 is one. Otherwise, a privileged-operation exception is recognized. In the supervisor state, any value for the secondary-space access key is valid.

The priority of the recognition of exceptions and condition codes is shown in Figure 10-21 on page 10-58.

### Resulting Condition Code:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True length less than or equal to 256</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>True length greater than 256</td>
</tr>
</tbody>
</table>

### Program Exceptions:

- Access (fetch, primary virtual address, operand 2, MVCS; fetch, secondary virtual address, operand 2, MVCP; store, secondary virtual address, operand 1, MVCS; store, primary virtual address, operand 1, MVCP)
- Privileged operation (selected PSW-key-mask bit is zero in the problem state)
- Special operation
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.

7.A Access exceptions for second and third instruction halfwords.

7.B Special-operation exception due to the secondary-space control, bit 37 of control register 0, being zero, to DAT being off, or to the CPU being in the access-register or home-space mode.

8. Privileged-operation exception due to selected PSW-key-mask bit being zero in the problem state.

9. Completion due to length zero.

10. Access exceptions for operands.

Figure 10-21. Priority of Execution: MOVE TO PRIMARY and MOVE TO SECONDARY

Programming Notes:

1. MOVE TO PRIMARY and MOVE TO SECONDARY can be used in a loop to move a variable number of bytes of any length. See the programming note under MOVE WITH KEY.

2. MOVE TO PRIMARY and MOVE TO SECONDARY should be used only when movement is between different address spaces. The performance of these instructions on most models may be significantly slower than that of MOVE WITH KEY, MOVE (MVC), or MOVE LONG. In addition, the definition of overlapping operands for MOVE TO PRIMARY and MOVE TO SECONDARY is not compatible with the more precise definitions for MOVE (MVC), MOVE WITH KEY, and MOVE LONG.

MOVE WITH DESTINATION KEY

\[ MVCDK \quad D_1(B_1), D_2(B_2) \quad [SSE] \]

<table>
<thead>
<tr>
<th>'E50F'</th>
<th>B_1</th>
<th>D_1</th>
<th>B_2</th>
<th>D_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The first operand is replaced by the second operand. The accesses to the destination-operand location are performed by using the key specified in general register 1, and the accesses to the source-operand location are performed by using the PSW key.

The first and second operands are of the same length, which is specified by bits 56-63 of general register 0. Bits 0-55 of general register 0 are ignored.

Bits 56-59 of general register 1 are used as the specified access key. Bits 0-55 and 60-63 of general register 1 are ignored.

The contents of general registers 0 and 1 are as follows:

```
GR0

|-----------------|   |
|                | L |
| 0              | 56 63 |
```

```
GR1

|-----------------| Key |
|                |///|
| 0              | 56 60 63 |
```

L specifies the number of bytes to the right of the first byte of each operand. Therefore, the length in bytes of each operand is 1-256, corresponding to a length code in L of 0-255.

The fetch accesses to the second-operand location are performed by using the PSW key, and the store accesses to the first-operand location are performed by using the key specified in general register 1.

Each of the operands is processed left to right. When the operands overlap destructively in real storage, the results in the first-operand location are unpredictable. Except for this unpredictability in the case of destructive overlap, the storage-operand-consistency rules are the same as for the MOVE (MVC) instruction.

Special Conditions

In the problem state, the operation is performed only if the access key specified in general register 1 is valid, that is, if the corresponding PSW-key-mask bit in control register 3 is one. Otherwise, a privileged-operation exception is recognized. In the supervisor state, any value for the specified access key is valid.

Condition Code: The code remains unchanged.
Program Exceptions:
- Access (fetch, operand 2; store, operand 1)
- Privileged operation (selected PSW-key-mask bit is zero in the problem state)

Programming Note: See the programming notes for the MOVE WITH SOURCE KEY instruction.

MOVE WITH KEY

\[
\text{MVCK } D_{basesuperone}(R_{basesuperone}), D_{basesupertwo}(B_{basesupertwo}), R_{basesuperthree} \quad [SS]
\]

<table>
<thead>
<tr>
<th>'09'</th>
<th>R1</th>
<th>R3</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

The first operand is replaced by the second operand. The fetch accesses to the second-operand location are performed by using the key specified in the third operand, and the store accesses to the first-operand location are performed by using the PSW key.

Bit positions 56-59 of general register \( R_3 \) are used as the source access key. Bit positions 0-55 and 60-63 of the register are ignored.

General register \( R_1 \) contains an unsigned binary integer called the true length. In the 24-bit or 31-bit addressing mode, the true length is in bit positions 32-63 of the register, and the contents of bit positions 0-31 of the register are ignored. In the 64-bit addressing mode, the true length is in bit positions 0-63 of the register.

The contents of the general registers just described are as follows:

24-Bit or 31-Bit Addressing Mode

\[
\begin{array}{|c|}
\hline
R_1 \\
\hline
0 & 32 & 63 \\
\hline
\end{array}
\]

True Length

64-Bit Addressing Mode

\[
\begin{array}{|c|}
\hline
R_1 \\
\hline
0 & 63 \\
\hline
\end{array}
\]

True Length

\[
\begin{array}{|c|}
\hline
R_3 \\
\hline
0 & 56 & 60 & 63 \\
\hline
\end{array}
\]

Key

The first and second operands are of the same length, called the effective length. The effective length is equal to the true length or 256, whichever is less. Access exceptions for the first and second operands are recognized only for that portion of the operand within the effective length. When the effective length is zero, no access exceptions are recognized for the first and second operands, and no movement takes place.

Each storage operand is processed left to right. When the storage operands overlap, the result is obtained as if the operands were processed one byte at a time and each result byte were stored immediately after the necessary operand byte was fetched. The storage-operand-consistency rules are the same as for the MOVE (MVC) instruction.

As part of the execution of the instruction, the value of the true length is used to set the condition code. If the true length is 256 or less, including zero, the true length and effective length are equal, and condition code 0 is set. If the true length is greater than 256, the effective length is 256, and condition code 3 is set.

Special Conditions

In the problem state, the operation is performed only if the source access key is valid, that is, if the corresponding PSW-key-mask bit in control register 3 is one. Otherwise, a privileged-operation exception is recognized. In the supervisor state, any value for the source access key is valid.

The priority of the recognition of exceptions and condition codes is shown in Figure 10-22 on page 10-60.

Resulting Condition Code:

0  True length less than or equal to 256
1  --
2  --
3  True length greater than 256

Program Exceptions:
- Access (fetch, operand 2; store, operand 1)
- Privileged operation (selected PSW-key-mask bit is zero in the problem state)
1. Exceptions with the same priority as the priority of program-interruption conditions for the general case.

7. Access exceptions for second and third instruction halfwords.

8. Privileged-operation exception due to selected PSW-key-mask bit being zero in the problem state.

9. Completion due to length zero.

10. Access exceptions for operands.

Figure 10-22. Priority of Execution: MOVE WITH KEY

Programming Notes:

1. MOVE WITH KEY can be used in a loop to move a variable number of bytes of any length, as follows:

   LOOP MVCK D1(R1,B1),D2(B2),R3
   BC 8,END
   AHI B1,256
   AHI B2,256
   AHI R1,-256
   B LOOP
   END [Any instruction]

   The above program is for execution in the 24-bit or 31-bit addressing mode. In the 64-bit addressing mode, AGHI instructions should be substituted for the AHI instructions.

2. The performance of MOVE WITH KEY on most models may be significantly slower than that of the MOVE (MVC) and MOVE LONG instructions. Therefore, MOVE WITH KEY should not be used if the keys of the source and the target are the same.

MOVE WITH SOURCE KEY

   MVCSK D1(B1),D2(B2) [SSE]

   'E50E' B1 D1 B2 D2

   The first operand is replaced by the second operand. The accesses to the source-operand location are performed by using the key specified in general register 1, and the accesses to the destination-operand location are performed by using the PSW key.

   The first and second operands are of the same length, which is specified by bits 56-63 of general register 0. Bits 0-55 of general register 0 are ignored.

   Bits 56-59 of general register 1 are used as the specified access key. Bits 0-55 and 60-63 of general register 1 are ignored.

   The contents of general registers 0 and 1 are as follows:

   | GR0 | L |
   |-----------------|
   | 0  | 56 63 |

   L specifies the number of bytes to the right of the first byte of each operand. Therefore, the length in bytes of each operand is 1-256, corresponding to a length code in L of 0-255.

   The fetch accesses to the second-operand location are performed by using the key specified in general register 1, and the store accesses to the first-operand location are performed by using the PSW key.

   Each of the operands is processed left to right. When the operands overlap destructively in real storage, the results in the first-operand location are unpredictable. Except for this unpredictability in the case of destructive overlap, the storage-operand-consistency rules are the same as for the MOVE (MVC) instruction.

Special Conditions

In the problem state, the operation is performed only if the access key specified in general register 1 is valid, that is, if the corresponding PSW-key-mask bit in control register 3 is one. Otherwise, a privileged-operation exception is recognized. In the supervisor state, any value for the specified access key is valid.

Condition Code: The code remains unchanged.
Program Exceptions:
- Access (fetch, operand 2; store, operand 1)
- Privileged operation (selected PSW-key-mask bit is zero in the problem state)

Programming Notes:
1. When data is to be moved alternately in both directions between two storage areas that are fetch protected by means of different keys, then MOVE WITH SOURCE KEY and MOVE WITH DESTINATION KEY can be used while leaving the PSW key unchanged; and this may be, on most models, significantly faster than using MOVE WITH KEY along with SET PSW KEY FROM ADDRESS to change the PSW key.

2. MOVE WITH SOURCE KEY and MOVE WITH DESTINATION KEY should be used only when movement is between storage areas having different keys. The performance of these instructions on most models may be significantly slower than that of the MOVE (MVC) instruction.

3. MOVE WITH SOURCE KEY or MOVE WITH DESTINATION KEY can be used in a loop to move a variable number of bytes as shown in the following example. In the example, the specified access key, the first-operand address, the second-operand address, and the length of each operand are assumed to be in general registers 1-4, respectively, at the beginning of the example. The length of each operand is treated as a 32-bit signed value, and a negative value is treated as zero.

```
LTR 4,4
BC 12,END
AHI 4,-256
BC 12,LAST
LA 0,255
LOOP MVCSK 0(2),0(3)
  LA 2,256(2)
  LA 3,256(3)
  AHI 4,-256
  BC 2,LOOP
LAST LA 0,255(4)
  MVCSK 0(2),0(3)
END [Any instruction]
```

A page-in operation is performed which transfers a 4K-byte block to the real-storage location designated by general register \( R_1 \) from the expanded-storage block designated by general register \( R_2 \).

Bits 32-63 of general register \( R_2 \) are a 32-bit unsigned binary integer called the expanded-storage-block number. This number designates the 4K-byte block of expanded storage which is to be transferred. If the expanded-storage-block number designates an inaccessible block in expanded storage, condition code 3 is set.

The contents of general register \( R_1 \) are a real address which designates a 4K-byte block in main storage. In the 24-bit-addressing mode, bits 40-51 designate the block, and bits 0-39 are ignored. In the 31-bit-addressing mode, bits 33-51 designate the block, and bits 0-32 are ignored. In the 64-bit-addressing mode, bits 0-51 designate the block. In all modes, bits 52-63 of the address are ignored.

Because it is a real address, the address designating the main-storage block is not subject to dynamic address translation. PAGE IN is not subject to key-controlled storage protection, but low-address protection does apply. PAGE IN is not subject to program-event recording for storage alteration.

A serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed.

If the page-in operation is completed with no errors, condition code 0 is set.

If the page-in operation encounters an expanded-storage data error, condition code 1 is set. For an expanded-storage data-error condition, the contents of the entire 4K-byte block in real storage is unpredictable, but this condition does not result in the generation of invalid checking-block codes in real storage.
If the expanded-storage block is not available, that is, the block is not provided or is not currently in the configuration, then condition code 3 is set, and no other action is taken.

**Operation of PAGE IN in a Multiple-CPU Configuration**

The accesses to main storage and to expanded storage by PAGE IN are not necessarily single-access references and are not necessarily performed in a left-to-right direction, as observed by other CPUs and by channel programs.

See also the description under PAGE OUT.

**Resulting Condition Code:**

- 0 Page-in operation completed
- 1 Expanded-storage data error
- 2 --
- 3 Expanded-storage block not available

**Program Exceptions:**

- Addressing (block designated by general register R1)
- Operation (if the expanded-storage facility is not installed)
- Privileged operation
- Protection (block designated by general register R1; low-address protection)

**PAGE OUT**

```
PGOUT R1, R2 [RRE]

'B2FF'  ///// / R1 R2
```

A page-out operation is performed which transfers a 4K-byte block from the real-storage location designated by general register R1 to the expanded-storage block designated by general register R2.

Bits 32-63 of general register R2 are a 32-bit unsigned binary integer called the expanded-storage-block number. This number designates the 4K-byte block of expanded storage which is to be replaced. If the expanded-storage-block number designates an inaccessible block in expanded storage, condition code 3 is set.

The contents of general register R1 are a real address which designates a 4K-byte block in main storage. In the 24-bit-addressing mode, bits 40-51 designate the block, and bits 0-39 are ignored. In the 31-bit-addressing mode, bits 33-51 designate the block, and bits 0-32 is ignored. In the 64-bit-addressing mode, bits 0-51 designate the block. In all modes, bits 52-63 of the address are ignored.

Because it is a real address, the address designating the main-storage block is not subject to dynamic address translation. PAGE OUT is not subject to key-controlled protection.

A serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed.

Depending on the model, after the data has been written to the expanded-storage block, a read-back-check operation may be performed to determine whether the data was written correctly. If the read-back-check operation determines that the data has been written correctly, condition code 0 is set. If the read-back-check operation encounters an expanded-storage data error, condition code 1 is set.

Most models do not perform the read-back-check operation, and, after the page-out operation is completed, condition code 0 is set.

Regardless of whether condition code 0 or condition code 1 is set, the entire 4K-byte block is written. Errors, if any, in the block after the block is written are preserved. Thus, if a subsequent execution of PAGE IN addresses the same expanded-storage block, the expanded-storage data error will be detected and condition code 1 will be indicated.

If the expanded-storage block is not available, that is, the block is not provided or is not currently in the configuration, then condition code 3 is set, and no other action is taken.

**Operation of PAGE OUT in a Multiple-CPU Configuration**

The accesses to main storage and to expanded storage by PAGE OUT are not necessarily single-access references and are not necessarily per-
formed in a left-to-right direction, as observed by other CPUs and by channel programs.

If two or more CPUs issue PAGE IN or PAGE OUT instructions at approximately the same instant in time, depending on the model, the operations may be performed one at a time, or the operations may be performed concurrently. Concurrent operation may occur even if the instructions address the same expanded-storage block.

When two or more PAGE OUT instructions addressing the same expanded-storage block are executed concurrently, the resulting values in the expanded-storage block for each group of bytes transferred may be from any of the instructions being executed simultaneously. The number of bytes transferred as a group depends on the model.

Similarly, for concurrent execution of a PAGE IN and a PAGE OUT instruction for the same expanded-storage block, the resulting values for each group of bytes transferred as a result of the execution of the PAGE IN instruction may be either the old or new values from the expanded-storage block.

Concurrent operation of paging instructions does not result in expanded-storage data errors.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Page-out operation completed</td>
</tr>
<tr>
<td>1</td>
<td>Expanded-storage data error</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>Expanded-storage block not available</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Addressing (block designated by general register R1)
- Operation (if the expanded-storage facility is not installed)
- Privileged operation

---

**PROGRAM CALL**

<table>
<thead>
<tr>
<th>PC</th>
<th>D2 (B2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>31</td>
</tr>
</tbody>
</table>

A program-call number specified by the second-operand address is used in a two-level or three-level lookup to locate an entry-table entry (ETE).

The program is authorized to use the ETE when the AND of the PSW-key mask in control register 3 and the authorization key mask in the ETE is nonzero or when the CPU is in the supervisor state.

When the PC-type bit, bit 128 of the ETE, is zero, an operation called basic PROGRAM CALL is performed. When the PC-type bit is one, an operation called stacking PROGRAM CALL is performed.

Basic PROGRAM CALL, in the 24-bit or 31-bit addressing mode, loads the basic-addressing-mode bit, bits 33-62 of the updated instruction address, and the problem-state bit from the PSW into bit positions 32-63 of general register 14, and it leaves bits 0-31 of this register unchanged. In the 64-bit addressing mode, bits 0-62 of the updated instruction address and the problem-state bit are placed in bit positions 0-63 of general register 14. In any addressing mode, the PSW-key mask and PASN are placed in bit positions 32-63 of general register 3, and bits 0-31 of this register remain unchanged.

Stacking PROGRAM CALL places the entire PSW contents, except with an unpredictable PER mask, and also the PSW-key mask, PASN, SASN, and EAX in a linkage-stack program-call state entry that it forms. A called-space identification, an indication of whether the resulting addressing mode is the 64-bit mode, the numeric part of the program-call number, and the contents of general registers 0-15 and access registers 0-15 also are placed in the state entry. If the ASN-and-LX-reuse facility is installed and enabled, the PASTEIN and SASTEIN also are placed in the state entry.

For basic PROGRAM CALL, the extended-addressing-mode bit, bit 31 of the PSW, must have the same value as the entry-extended-addressing-mode bit, bit 129 of the ETE;
otherwise, a special-operation exception is recognized. Basic PROGRAM CALL does not change bit 31 of the PSW and, therefore, does not switch between a basic addressing mode (the 24-bit or 31-bit mode) and the extended addressing mode (the 64-bit mode). In the 24-bit or 31-bit addressing mode, basic PROGRAM CALL sets the basic-addressing-mode bit, bit 32 of the PSW, with the value of the entry-basic-addressing-mode bit, bit 32 of the ETE, and, thus, it may switch between the 24-bit and 31-bit addressing modes. In the 64-bit addressing mode, bit 32 of the PSW remains unchanged.

Stacking PROGRAM CALL, when bit 129 of the ETE is zero, sets bit 31 of the PSW to zero and sets bit 32 of the PSW with the value of bit 32 of the ETE. When bit 129 of the ETE is one, stacking PROGRAM CALL sets bits 31 and 32 of the PSW to one. Thus, stacking PROGRAM CALL can set the 24-bit, 31-bit, or 64-bit addressing mode.

When the resulting addressing mode is the 24-bit or 31-bit mode, both basic and stacking PROGRAM CALL place bits 33-62 of the entry instruction address in the ETE, which are bits 33-62 of the ETE, with 33 leftmost and one rightmost zeros appended, in bit positions 64-127 of the PSW as the new instruction address, and they place the entry-problem-state bit, bit 63 of the ETE, in bit position 15 of the PSW as the new problem-state bit. Bits 32-63 of the entry parameter in the ETE are placed in bit positions 32-63 of general register 4, and bits 0-31 of this register remain unchanged.

When the resulting addressing mode is the 64-bit mode, both basic and stacking PROGRAM CALL place bits 0-62 of the entry instruction address, bits 0-62 of the ETE, with one rightmost zero appended, in bit positions 64-127 of the PSW, and they place bit 63 of the ETE in bit position 15 of the PSW. Bits 0-63 of the entry parameter in the ETE are placed in general register 4.

Basic PROGRAM CALL ORs the entry key mask from the ETE into the PSW-key mask in control register 3. Stacking PROGRAM CALL does the same, or it replaces the PSW-key mask with the entry key mask, as determined by the PSW-key-mask control in the ETE.

Stacking PROGRAM CALL optionally replaces the PSW key in the PSW and the EAX in control register 8 from the ETE, and it sets the address-space-control bits in the PSW, as determined by control bits in the ETE.

The ETE causes a space-switching operation to occur if it contains a nonzero ASN. When the ETE contains a zero ASN, the operation is called PROGRAM CALL to current primary (PC-cp); when the ETE contains a nonzero ASN, the operation is called PROGRAM CALL with space switching (PC-ss). When space switching is specified, the new PASN is loaded into control register 4 from the ETE, and a new primary-ASTE origin (PASTE0) is loaded into control register 5, also from the ETE. From the PASTE, a new primary ASCE (PASCE) and AX are loaded into control registers 1 and 4, respectively. If ASN-and-LX reuse is enabled, a new primary ASTE instance number (PASTEIN) is loaded into control register 4, also from the PASTE.

In both PC-cp and PC-ss, the SASN and secondary ASCE (SASCE) are set equal to the original PASN and PASCE, respectively, and, if ASN-and-LX reuse is enabled, the secondary ASTE instance number (SASTEIN) in control register 3 is set equal to the original PASTEIN. However, the space-switching stacking PROGRAM CALL operation may instead set the SASN and SASCE equal to the new PASN and PASCE, respectively, and, if ASN-and-LX reuse is enabled, set the SASTEIN equal to the new PASTEIN, as determined by a control bit in the ETE.

In a PC-ss to the base space of the dispatchable unit when the dispatchable unit is subspace active, bits 0-55 and 58-63 of the new PASCE are replaced by the same bits of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. This occurs before the possible setting of the SASCE equal to the PASCE.

**PROGRAM CALL PC-Number Translation**

The second-operand address is not used to address data; instead, the rightmost 20 or 32 bits of the address are used as a PC number divided into two fields: a linkage index (LX) and an entry index (EX). The entry index is always the rightmost eight bits of the PC number. When the ASN-and-LX-reuse is not installed or is not
enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the PC number is 20 bits, and the leftmost 12 bits of the number are the linkage index. In this case, the second-operand address has the following format:

Second-operand Address when ASN-and-LX Reuse Is Not Enabled

```
+--------+--------+
| LX     | EX     |
+--------+--------+
```

Bit 44 of the effective address has no special meaning and may be zero or one.

When the ASN-and-LX-reuse facility is installed and enabled, referred to simply by saying that ASN-and-LX reuse is enabled, the linkage index is further divided into a linkage first index (LFX) and a linkage second index (LSX). The linkage second index is always the five bits immediately on the left of the entry index. The size and format of the linkage first index depend on whether the PC number is 20 bits or 32 bits, which in turn depends on whether bit 44 of the second-operand address is zero or one, respectively. In these cases, the second-operand address has the following formats:

Second-Operand Address when ASN-and-LX Reuse Is Enabled and Bit 44 Is Zero

```
+--------+--------+--------+--------+
| LX     | LFX    | LSX    | EX     |
+--------+--------+--------+--------+
```

Second-Operand Address when ASN-and-LX Reuse Is Enabled and Bit 44 Is One

```
+--------+--------+--------+--------+--------+
| LX     | LFX1   | LFX2   | LSX    | EX     |
+--------+--------+--------+--------+--------+
```

When ASN-and-LX reuse is enabled and bit 44 of the second-operand address is zero, the PC number is 20 bits, and the linkage first index is bits 44-50, or bits 45-50 (LFX2) with a zero appended on the left. Thus, the linkage first index is seven bits of which the leftmost bit is always zero. When bit 44 is one, the PC number is 32 bits, and the linkage first index is bits 45-50 (LFX2) with bits 32-43 (LFX1) appended on the left, or 18 bits, and bit 44 is not a numeric part of the PC number. However, a linkage first table can contain at most 16,384 entries, and, therefore, the leftmost four bits of the linkage first index, bits 32-35 of the second-operand address, must always be zeros; otherwise, an LFX-translation exception is recognized.

**Linkage Index (LX):** When ASN-and-LX reuse is not enabled, bits 44-55 of the second-operand address are the linkage index and are used to select an entry from the linkage table designated by the linkage-table designation in the primary ASTE.

**Linkage First Index (LFX):** When ASN-and-LX reuse is enabled and bit 44 of the second-operand address is zero, bits 44-50 of the second-operand address are the linkage first index. When ASN-and-LX reuse is enabled and bit 44 of the second-operand address is one, bits 45-50 with bits 32-43 appended on the left are the linkage first index. In either case, the linkage first index is used to select an entry from the linkage first table designated by the linkage-first-table designation in the primary ASTE.

**Linkage Second Index (LSX):** When ASN-and-LX reuse is enabled, bits 51-55 of the second-operand address are the linkage second index and are used to select an entry from the linkage second table designated by the linkage-second-table entry.

**Entry Index (EX):** Bits 56-63 of the second-operand address are the entry index and are used to select an entry from the entry table designated by the linkage-table entry or the linkage-second-table entry.

When the PC number is 20 bits, bits 32-43 of the second-operand address are ignored. Bits 0-31 of the second-operand address are always ignored.

When ASN-and-LX reuse is enabled, the second word of the linkage-second-table entry used contains a linkage-second-table-entry sequence number (LSTESN). When this LSTESN is nonzero, it must be equal to an LSTESN specified in bit positions 0-31 of general register 15; otherwise, an LSTE-sequence exception is recognized. When ASN-and-LX reuse is not enabled (including when it is not installed) or the LSTESN in the linkage-second-table entry is zero, bits 0-31 of general register 15 are ignored.
The 32-byte entry-table entry (ETE) has the following format:

When Resulting Addressing Mode Is the 24-Bit or 31-Bit Mode

<table>
<thead>
<tr>
<th>A</th>
<th>EIA</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
<td>63</td>
</tr>
</tbody>
</table>

When Resulting Addressing Mode Is the 64-Bit Mode

<table>
<thead>
<tr>
<th>EIA</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AKM</th>
<th>ASN</th>
<th>EKM</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>80</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>112</td>
<td>127</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cntrl/EK</th>
<th>EEAX</th>
<th>ASTE Adr</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>144</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>186</td>
<td>191</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>192</td>
</tr>
</tbody>
</table>

Bits 128-143 of the ETE have the following detailed format:

<table>
<thead>
<tr>
<th>T</th>
<th>G</th>
<th>K</th>
<th>M</th>
<th>E</th>
<th>C</th>
<th>S</th>
<th>EK</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>131</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
</tr>
</tbody>
</table>

For basic PROGRAM CALL in the 24-bit or 31-bit addressing mode when bit 32 of the ETE (A) is zero (specifying the 24-bit mode), and for stacking PROGRAM CALL when bits 32 (A) and 129 (G) are zeros (specifying the 24-bit mode), bits 33-39 must be zeros; otherwise, a PC-translation-specification exception is recognized.

After the ETE has been fetched, if the current PSW specifies the problem state, the current PSW-key mask in control register 3 is tested against the AKM field in the ETE to determine whether the program is authorized to access this entry. The AKM and PSW-key mask are ANDed, and, if the result is zero, a privileged-operation exception is recognized. The PSW-key mask in control register 3 remains unchanged. When PROGRAM CALL is executed in the supervisor state, the AKM field is ignored.

If the result of the AND of the AKM and the PSW-key mask is not zero, or if the CPU is in the supervisor state, the execution of the instruction continues.

If bit 128 of the ETE (T) is zero, the basic PROGRAM CALL operation is specified. If bit 128 of the ETE is one, the stacking PROGRAM CALL operation is specified.

**Basic PROGRAM CALL**

The following operations are performed when basic PROGRAM CALL is specified.

Bit 31 of the current PSW (the extended-addressing-mode bit) must equal bit 129 (G) of the ETE; otherwise, a special-operation exception is recognized.

In the 24-bit or 31-bit addressing mode, bits 97-126 of the PSW (bits 33-62 of the updated instruction address) are placed in bit positions 33-62 of general register 14, bit 32 of the PSW (the basic-addressing-mode bit) is placed in bit position 32 of the register, and bit 15 of the PSW (the problem-state bit) is placed in bit position 63 of the register. Bits 0-31 of the register remain unchanged.

In the 64-bit addressing mode, bits 64-126 of the PSW (bits 0-62 of the updated instruction address) are placed in bit positions 0-62 of general register 14, and bit 15 of the PSW (the problem-state bit) is placed in bit position 63 of the register.

In the 4-bit or 31-bit addressing mode, bits 32 and 33-62 of the ETE (A and the EIA), with a zero appended on the right of bits 33-62, are placed in PSW bit positions 32 and 97-127, respectively (the basic-addressing-mode bit and bits 33-63 of the instruction address). In the 64-bit addressing mode, bits 0-62 of the ETE, with a zero appended on the right, are placed in PSW bit positions 64-127 (the instruction address), and PSW bit 32 remains unchanged. In any addressing mode, bit 63 of the ETE (P) is placed in PSW bit position 15 (the problem-state bit).
The PSW-key mask, bits 32-47 of control register 3, is placed in bit positions 32-47 of general register 3, and the current PASN, bits 48-63 of control register 4, is placed in bit positions 48-63 of general register 3. Bits 0-31 of general register 3 remain unchanged.

Bits 96-111 of the ETE (the EKM) are ORed with the PSW-key mask, bits 32-47 of control register 3, and the result replaces the PSW-key mask in control register 3.

In the 24-bit or 31-bit addressing mode, bits 224-255 of the ETE (bits 32-63 of the entry parameter) are loaded into bit positions 32-63 of general register 4, and bits 0-31 of the register remain unchanged. In the 64-bit addressing mode, bits 192-255 of the ETE (the entry parameter), are loaded into bit positions 0-63 of general register 4.

Stacking PROGRAM CALL

The following operations are performed when stacking PROGRAM CALL is specified.

The stacking process is performed to form a linkage-stack program-call state entry and place the following information in the state entry: current PSW (with an unpredictable PER mask), PSW-key mask, PASN, SASN, EAX, called-space identification, an indication of whether the resulting addressing mode is the 64-bit mode, numeric part of the program-call number, contents of general registers 0-15, and contents of access registers 0-15, and, if ASN-and-LX reuse is enabled, PASTEIN and SASTEIN. This is described in "Stacking Process" on page 5-83. The entry-type code in the state entry is 0001101 binary.

When bit 129 of the ETE (G) is zero, bit 31 of the PSW (the extended-addressing-mode bit) is set to zero, and bit 32 of the ETE (A) is placed in bit position 32 of the PSW (the basic-addressing-mode bit). (The addressing mode is set to the 24-bit mode if bit 32 is zero or to the 31-bit mode if bit 32 is one.) When bit 129 of the ETE is one, bits 31 and 32 of the PSW are set to one. (The 64-bit addressing mode is set.)

When the resulting addressing mode is the 24-bit or 31-bit mode, bits 33-62 of the ETE (the EIA), with 33 leftmost and one rightmost zeros appended, are placed in PSW bit positions 64-127 (the instruction address). When the resulting addressing mode is the 64-bit mode, bits 0-62 of the ETE (the EIA), with one rightmost zero appended, are placed in PSW bit positions 64-127.

Bit 63 of the ETE (P) is placed in PSW bit position 15 (the problem-state bit).

When bit 131 of the ETE (K) is zero, bits 8-11 of the PSW (the PSW key) remain unchanged. When bit 131 of the ETE is one, bits 136-139 of the ETE (the EK) replace the PSW key in the PSW.

When bit 132 of the ETE (M) is zero, bits 96-111 of the ETE (the EKM) are ORed with the PSW-key mask, bits 32-47 of control register 3, and the result replaces the PSW-key mask in control register 3. When bit 132 of the ETE is one, bits 96-111 of the ETE replace the PSW-key mask in control register 3.

When bit 133 of the ETE (E) is zero, the EAX, bits 32-47 of control register 8, remains unchanged. When bit 133 of the ETE is one, bits 144-159 of the ETE (the EEAX) replace the EAX in control register 8.

When bit 134 of the ETE (C) is zero, bits 16 and 17 of the PSW (the address-space-control bits) are set to 00 binary (primary-space mode). When bit 134 of the ETE is one, the address-space-control bits in the PSW are set to 01 binary (access-register mode).

When the resulting addressing mode is the 24-bit or 31-bit mode, bits 224-255 of the ETE (bits 32-63 of the entry parameter) are loaded into bit positions 32-63 of general register 4, and bits 0-31 of this register remain unchanged. When the resulting addressing mode is the 64-bit mode, bits 192-255 of the ETE (the entry parameter), are loaded into bit positions 0-63 of general register 4.

Key-controlled protection does not apply to references to the linkage stack, but low-address and page protection do apply.

PROGRAM CALL to Current Primary (PC-cp)

If bits 80-95 of the ETE (the ASN), are zeros, PROGRAM CALL to current primary (PC-cp) is specified, and the execution of the instruction is
completed after the operations described in “PROGRAM CALL PC-Number Translation” and either “Basic PROGRAM CALL” or “Stacking PROGRAM CALL” have been performed and the following operations have been performed.

The current PASN, bits 48-63 of control register 4, is placed in bit positions 48-63 of control register 3 to become the current SASN.

The current PASCE in control register 1 is placed in control register 7 to become the current SASCE.

If ASN-and-LX reuse is enabled, the current PASTEIN, bits 0-31 of control register 4, is placed in bit positions 0-31 of control register 3 to become the current SASTEIN.

The basic PC-cp operation is depicted in parts 1-3 of Figure 10-24 on page 10-72. The stacking PC-cp operation is depicted in parts 1, 4, and 5 of the figure.

PROGRAM CALL with Space Switching (PC-ss)

If the ASN in the ETE is nonzero, PROGRAM CALL with space switching (PC-ss) is specified, and the execution of the instruction is completed after the operations described in “PROGRAM CALL PC-Number Translation” and either “Basic PROGRAM CALL” or “Stacking PROGRAM CALL” have been performed and the following operations have been performed.

Bits 80-95 of the ETE (the ASN) are placed in bit positions 48-63 of control register 4 as the new PASN.

Bits 161-185 of the ETE, with six zeros appended on the right, are used as the real address of the ASTE designated by the new PASN. An ASX-translation exception is recognized if bit 0 of the ASTE is one.

Bits 64-127 of the ASTE (the ASCE) are placed in control register 1 as the new PASCE.

Bits 32-47 of the ASTE (the AX) are placed in bit positions 32-47 of control register 4 as the new authorization index.

If ASN-and-LX reuse is enabled, bits 352-383 of the ASTE (the ASTEIN) are placed in bit positions 0-31 of control register 3 as the new PASTEIN.

Bits 33-57 of the ASTE address are placed in bit positions 33-57 of control register 5 as the new primary-ASTE origin, and zeros are placed in bit positions 32 and 58-63. Bits 0-31 of the register remain unchanged.

In basic PROGRAM CALL, or in stacking PROGRAM CALL when bit 135 of the ETE (S) is zero, the PASN existing before the PASN is replaced from the ETE is placed in bit positions 48-63 of control register 3 to become the current SASN, and the PASCE existing before the PASCE is replaced from the ASTE is placed in control register 7 to become the current SASCE. If ASN-and-LX reuse is enabled, the PASTEIN existing before the PASTEIN is replaced from the ASTE is placed in bit positions 0-31 of control register 3 to become the current SASTEIN. (The SASN and SASCE are set equal to the old PASN and PASCE, respectively, and, if ASN-and-LX reuse is enabled, the SASTEIN is set equal to the old PASTEIN.)

In stacking PROGRAM CALL when bit 135 of the ETE (S) is one, the SASN is replaced by the PASN after the PASN is replaced from the ETE, and the SASCE is replaced by the PASCE after the PASCE is replaced from the ASTE. If ASN-and-LX reuse is enabled, the SASTEIN is replaced by the PASTEIN after the PASTEIN is replaced from the ASTE. (The SASN and SASCE are set equal to the new PASN and PASCE, respectively, and, if ASN-and-LX reuse is enabled, the SASTEIN is set equal to the new PASTEIN.)

The description in this paragraph applies to use of the subspace-group facility. After the new PASCE has been placed in control register 1 and the new primary-ASTE origin has been placed in control register 5, if (1) the subspace-group-control bit, bit 54, in the PASCE is one, (2) the dispatchable unit is subspace active, and (3) the primary-ASTE origin designates the ASTE for the base space of the dispatchable unit, then bits 0-55 and 58-63 of the PASCE are replaced by the same bits of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. This replacement occurs before a replacement of the SASCE in control register 7 by the PASCE. Further details are in “Subspace-Replacement Operations” on page 5-68.
The PC-ss operation is depicted in parts 1 and 4-6 of Figure 10-24 on page 10-72.

**PROGRAM CALL Serialization**

For both the PC-cp and PC-ss operations, a serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. However, it is unpredictable whether or not a store into a trace-table entry or linkage-stack entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.

**Special Conditions**

The basic PROGRAM CALL operation can be performed successfully only when (1) the CPU is in the primary-space mode at the beginning of the operation, (2) the subsystem-linkage control, bit 0 of the linkage-table designation or linkage-first-table designation in the current primary ASN-second-table entry, is one, and (3) the extended-addressing-mode bit, bit 31 of the current PSW, equals the entry-extended-addressing-mode bit, bit 129 of the entry-table entry. Stacking PROGRAM CALL can be performed successfully only when the CPU is in the primary-space mode or access-register mode at the beginning of the operation and the subsystem-linkage control is one. In addition, PC-ss can be performed successfully only when the ASN-translation control, bit 44 of control register 14, is one. If any of these rules is violated, a special-operation exception is recognized.

When ASN-and-LX reuse is enabled and the LSTESN in the linkage-second-table entry is nonzero, that LSTESN must be equal to the LSTESN specified in bit positions 0-31 of general register 15; otherwise, an LSTE-sequence exception is recognized.

A stack-full or stack-specification exception may be recognized during the stacking process.

When, for PC-ss, the primary space-switch-event-control bit, bit 57 of control register 1, is one either before or after the execution of the instruction, a space-switch-event program interruption occurs after the operation is completed. A space-switch-event program interruption also occurs after the completion of a PC-ss operation if a PER event is reported.

The operation is suppressed on all addressing and protection exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-23 on page 10-70.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch or store, except for key-controlled protection, linkage-stack entry)
- Addressing (linkage-table or linkage-first-table designation in primary ASN-second-table entry; linkage-table entry; linkage-first-table entry; linkage-second-table entry, entry-table entry; ASN-second-table entry, PC-ss only)
- ASX translation (PC-ss only)
- EX translation
- LFX translation
- LSTE sequence
- LSX translation
- LX translation
- PC-translation specification
- Privileged operation (AND of AKM and PSW-key mask is zero in the problem state)
- Space-switch event (PC-ss only)
- Special operation
- Stack full (stacking PC only)
- Stack specification (stacking PC only)
- Subspace replacement (PC-ss only)
- Trace

**Programming Note:** The effective address from which a PC number is derived is subject to the addressing mode in the current PSW. Therefore, bits 0-39 of the effective address in the 24-bit addressing mode, and bits 0-32 in the 31-bit addressing mode, are treated as containing zeros.
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Special-operation exception due to DAT being off or the CPU being in secondary-space mode or home-space mode.

8.A Trace exceptions.

8.B.1 Addressing exception for access to linkage-table designation or linkage-first-table designation in primary ASN-second-table entry.

8.B.2 Special-operation exception due to subsystem-linkage control in linkage-table designation or linkage-first-table designation being zero.

**Note:** The LFX-translation, LSX-translation, and LSTE-sequence exceptions can be recognized only if ASN-and-LX reuse is enabled, and the LX-translation exception cannot be recognized in that case.

8.B.3 LX-translation or LFX-translation exception due to linkage-table or linkage-first-table entry, respectively, being outside table.

8.B.4 Addressing exception for access to linkage-table or linkage-first-table entry.

8.B.5 LX-translation or LFX-translation exception due to I bit (bit 0) in linkage-table or linkage-first-table entry, respectively, being one.

8.B.6 Addressing exception for access to linkage-second-table entry.

8.B.7 LSX-translation exception due to I bit (bit 0) in linkage-second-table entry being one.

8.B.8 LSTE-sequence exception due to LSTE sequence number in linkage-second-table entry being nonzero and not equal to bits 0-31 of general register 15.

8.B.9 EX-translation exception due to entry-table entry being outside table.

8.B.10 Addressing exception for access to entry-table entry.

8.B.11 Special-operation exception due to the CPU being in access-register mode or extended-addressing-mode bit, bit 31 of PSW, not being equal to entry-extended-addressing-mode bit, bit 129 of entry-table entry (basic PC only).

*Figure 10-23 (Part 1 of 2). Priority of Execution: PROGRAM CALL*
| 8.B.12 | PC-translation-specification exception due to invalid combination (bits 33-39 not zeros when resulting addressing mode is 24 bit) in entry-table entry. |
| 8.B.13 | Privileged-operation exception due to zero result from ANDing PSW-key mask and AKM in the problem state. |
| 8.B.14 | Special-operation exception due to ASN-translation control, bit 44 of control register 14, being zero (PC-ss only). |
| 8.B.15 | Addressing exception for access to ASN-second-table entry (PC-ss only). |
| 8.B.16 | ASX-translation exception due to I bit (bit 0) in ASN-second-table entry being one (PC-ss only). |

**Note:** Subspace-replacement exceptions, which are not shown in detail in this figure, can occur with any priority after 8.B.16 and before 9.

| 8.B.17 | Access exceptions (fetch) for entry descriptor of the current linkage-stack entry (stacking PC only). |

**Note:** Exceptions 8.B.18-8.B.23 can occur only if there is not enough remaining free space in the current linkage-stack section.

| 8.B.18 | Stack-specification exception due to remaining-free-space value in current linkage-stack entry not being a multiple of 8. |
| 8.B.19 | Access exceptions (fetch) for second word of the trailer entry of the current section. The entry is presumed to be a trailer entry; its entry-type field is not examined (stacking PC only). |
| 8.B.20 | Stack-full exception due to forward-section validity bit in the trailer entry being zero (stacking PC only). |
| 8.B.21 | Access exceptions (fetch) for entry descriptor of the header entry of the next section (stacking PC only). This entry is presumed to be a header entry; its entry-type field is not examined. |
| 8.B.22 | Stack-specification exception due to not enough remaining free space in the next section (stacking PC only). |
| 8.B.23 | Access exceptions (store) for second word of the header entry of the next section. If there is no exception, the header is now called the current entry. |
| 8.B.24 | Access exceptions (store) for entry descriptor of the current entry and for the new state entry (stacking PC only). |
| 9. | Space-switch event (PC-ss only). |

*Figure 10-23 (Part 2 of 2). Priority of Execution: PROGRAM CALL*
### Basic PC-cp and PC-ss in 24- or 31-Bit Addressing Mode

**Entry-Table Entry**

<table>
<thead>
<tr>
<th>A</th>
<th>EIA</th>
<th>P</th>
<th>AKM</th>
<th>ASN</th>
<th>EKM</th>
<th>T</th>
<th>G</th>
<th>ASTE Addr.</th>
<th>EP 32-63</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Shown again below
- With 33 Zeros on Left

**PSW after**

<table>
<thead>
<tr>
<th>P</th>
<th>E</th>
<th>A</th>
</tr>
</thead>
</table>

- $E = 0$

**From above**

- AKM
- ASN
- EKM

**CR1 before**

- PASCE

**CR7 after**

- SASCE

**AND**

- Exception if Zero in Problem State

**CR3 32-63 before**

<table>
<thead>
<tr>
<th>PKM</th>
<th>SASN</th>
</tr>
</thead>
</table>

**CR3 32-63 after**

| PKM | SASN |

**CR4 32-63 before**

<table>
<thead>
<tr>
<th>AX</th>
<th>PASN</th>
</tr>
</thead>
</table>

**CR4 32-63 after**

| AX | PASN |

**GR3 32-63 after**

| PKM | PASN |

**PSW before**

<table>
<thead>
<tr>
<th>P</th>
<th>E</th>
<th>A</th>
</tr>
</thead>
</table>

- $E = 0$

- IA 33-62

**GR14 32-63 after**

| PKM | IA 33-62 | P |

---

*Figure 10-24 (Part 1 of 5). Execution of PROGRAM CALL*
Basic PC-cp and PC-ss in 64-Bit Addressing Mode

Figure 10-24 (Part 2 of 5). Execution of PROGRAM CALL
Stacking PC-cp and PC-ss from 24- or 31-Bit Addressing Mode to 64-Bit Mode

Entry-Table Entry

```
| EIA | P | AKM | ASN | EKM | T | G | K | M | E | C | S | EK | EEAX | ASTE Adr. | EP |
```

- Shown again below
- T=1
- G=1

```
PSW after Key P 0 C E A IA 0-62 0
```

From above

```
AKM ASN EKM
```

Privileged-Op. Exception if Zero in Problem State

```
AND
```

```
CR3 32-63 before PKM SASN
```

```
CR4 32-63 before AX PASN
```

```
CR8 32-63 before EAX
```

```
CR7 after SASCE
```

```
CR after
```

```
PC-cp, or PC-ss and S=0 *
```

```
PSW before PSW LS
```

*: If PC-ss and S=1, SASN is replaced by new PASN, SASCE is replaced by new PASCE, and, if ASN-and-LX reuse is enabled, SASTEIN is replaced by new PASTEIN.

*Figure 10-24 (Part 3 of 5). Execution of PROGRAM CALL*
### Stacking PC-cp and PC-ss from 64-Bit to 24- or 31-Bit Addressing Mode

#### Entry-Table Entry

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td>G</td>
<td>K</td>
<td>M</td>
<td>E</td>
<td>C</td>
<td>S</td>
<td>EEAX</td>
<td>ASTE Adr.</td>
<td>EP 32-63</td>
</tr>
</tbody>
</table>

-T=1
G=0
With 33 Zeros on Left
K=1
PSW after
Key: PC EIA
IA 0-62 0

---

- Exception if
- Zero in Problem
- State

#### Figure 10-24 (Part 4 of 5). Execution of PROGRAM CALL

*: If PC-ss and S=1, SASN is replaced by new PASN, SASCE is replaced by new PASCE, and, if ASN-and-LX reuse is enabled, SASTEIN is replaced by new PASTEIN.
Operations on ASN-Second-Table Entry for PC-ss

Figure 10-24 (Part 5 of 5). Execution of PROGRAM CALL

PROGRAM RETURN

The PSW, except for the PER-mask bit, saved in the last linkage-stack state entry is restored as the current PSW. The PER mask in the current PSW remains unchanged. The contents of general registers 2-14 and access registers 2-14 also are restored from the state entry. When the entry-type code in the entry descriptor of the state entry is 0001101 binary, indicating a program-call state entry, (1) the primary ASN (PASN), secondary ASN (SASN), PSW-key mask (PKM), and extended authorization index (EAX) in the control registers also are restored from the state entry. and (2) if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the primary ASTE instance number (PASTEIN) and secondary ASTE instance number (SASTEIN) in the control registers also are restored from the state entry. When the entry-type code is 0001100 binary, indicating a branch state entry, the current PASN, SASN, PKM, EAX, PASTEIN, and SASTEIN remain unchanged.

The last state entry is located, and information in it is restored, as described in "Unstacking Process" on page 5-86. The state entry is logically deleted from the linkage stack, and the linkage-stack-entry address in control register 15 is replaced by the
address of the next preceding state or header entry. This also is described in "Unstacking Process."

When the state entry is a program-call state entry, it causes a space-switching operation to occur if it contains a PASN that is not equal to the current PASN. When the state entry contains a PASN that is equal to the current PASN, the operation is called PROGRAM RETURN to current primary (PR-cp); when the state entry contains a PASN that is not equal to the current PASN, the operation is called PROGRAM RETURN with space switching (PR-ss). PASN translation occurs in PR-ss. SASN translation and authorization may occur in either PR-cp or PR-ss. The terms PR-cp and PR-ss do not apply when the state entry is a branch state entry.

When the ASN-and-LX-reuse facility is installed and enabled and PASN or SASN translation occurs, the PASTEIN or SASTEIN, respectively, saved in the state entry is compared to the ASTEIN in the located ASTE.

Key-controlled protection does not apply to accesses to the linkage stack, but low-address and page protection do apply.

The sections “PASN Translation,” “SASN Translation,” “SASN Authorization,” and “PROGRAM RETURN Serialization” apply only when the unstacked state entry is a program-call state entry. The functions described in those sections are not performed when the state entry is a branch state entry.

The actions involving the PASTEIN and SASTEIN occur only when ASN-and-LX reuse is enabled by the ASN-and-LX reuse control in control register 0.

PASN Translation

If the new PASN is equal to the old PASN in bit positions 48-63 of control register 4, PASN translation is not performed, the PASTEIN in control register 4 remain as restored from the state entry, and the PASCE in control register 1 and primary-ASTE origin (PASTEO) in control register 5 are not changed. In this case, there is not a test of whether the new PASTEIN is equal to the old PASTEIN.

If the new PASN is not equal to the old PASN, the new PASN replaces the PASN in bit positions 48-63 of control register 4 and is translated to locate a 64-byte ASTE. The ASN table-lookup process is described in "ASN Translation" on page 3-24. The exceptions associated with ASN translation are collectively called ASN-translation exceptions. These exceptions and their priority are described in Chapter 6, "Interruptions."

Bits 32-47 of the ASTE are placed in bit positions 32-47 of control register 4 as the new AX. The PASN and PASTEIN in control register 4 remain as restored from the state entry.

The description in this paragraph applies to use of the subspace-group facility when PASN translation has occurred. If (1) the subspace-group-control bit, bit 54, in the new PASCE is one, (2) the dispatchable unit is subspace active, and (3) the new primary-ASTE origin designates the ASTE for the base space of the dispatchable unit, then bits 0-55 and 58-63 of the new PASCE in control register 1 are replaced by the same bits of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. This replacement occurs, in the case when the new SASN is equal to the new PASN, before the SASCE is set equal to the PASCE. Further details are in "Subspace-Replacement Operations" on page 5-68.

SASN Translation

If the new SASN is equal to the new PASN, the SASCE in control register 7 is set equal to the new PASCE in control register 1. The SASTEIN, PKM, and SASN in control register 3 remain as restored from the state entry. In this case, there
is not a test of whether the new SASTEIN is equal
to the new PASTEIN.

If the new SASN is not equal to the new PASN, the new SASN is translated to locate a 64-byte ASTE.

If ASN-and-LX reuse is enabled, the SASTEIN saved in bytes 176-179 of the state entry must equal the ASTEIN in bit positions 352-383 of the located ASTE; otherwise, an ASTE-instance exception is recognized.

Bits 64-127 of the ASTE are placed in control register 7 as the new SASCE.

Control register 3 remains as restored from the state entry.

**SASN Authorization**

If the new SASN is not equal to the new PASN, the authority-table origin (ATO) from the ASTE for the new SASN is used as the base for a third table lookup. The new authorization index, bits 32-47 of control register 4, is used, after it has been checked against the authority-table length, as the index to locate the entry in the authority table. The authority-table lookup is described in "ASN Authorization" on page 3-30.

The description in this paragraph applies to use of the subspace-group facility when SASN translation and authorization have occurred. If (1) the subspace-group-control bit, bit 54, in the new SASCE is one, (2) the dispatchable unit is subspace active, and (3) the ASTE origin obtained by SASN translation designates the ASTE for the base space of the dispatchable unit, then bits 0-55 and 58-63 are replaced by the same bits of the ASCE of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. Further details are in "Subspace-Replacement Operations" on page 5-68.

**PROGRAM RETURN Serialization**

When the unstacked state entry is a program-call state entry, a serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. However, it is unpredictable whether or not a store into a trace-table entry or linkage-stack entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.

**Special Conditions**

The instruction can be executed successfully only when the CPU is in the primary-space mode or access-register mode at the beginning of the operation. In addition, the ASN-translation process can be performed, for either the PASN or the SASN, only when the ASN-translation control, bit 44 of control register 14, is one. If either of these rules is violated, a special-operation exception is recognized.

A stack-empty, stack-operation, stack-specification, or stack-type exception may be recognized during the unstacking process.

If ASN-and-LX reuse is enabled, the restored PASTEIN must equal the ASTEIN in the located ASTE if PASN translation is performed, and the restored SASTEIN must equal the ASTEIN in the located ASTE if SASN translation is performed; otherwise, an ASTE-instance exception is recognized.

When, for PR-ss, the primary space-switch-event control, bit 57 of control register 1, is one either before or after the execution of the instruction, a space-switch-event program interruption occurs after the operation is completed. A space-switch-event program interruption also occurs after the completion of a PR-ss operation if a PER event is reported.

The PSW which is to be loaded by the instruction is not checked for validity before it is loaded. However, after loading, a specification exception is recognized, and a program interruption occurs, if any of bits 0, 2-4, 12, 24-30, and 33-63 of the PSW is a one, if bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros, if bits 31 and 32 are both zero and bits 64-103 are not all zeros, or if bits 31 and 32 are one and zero, respectively. In these cases, the operation is completed, and the resulting instruction-length code is 0. The specification exception, which in this case is listed as a program exception in this instruction, is described in "Early Exception Recognition" on page 6-9. It may be considered as occurring early in the process of preparing to execute the following instruction.
If a space-switch event is indicated and the PSW that was loaded by the instruction is invalid because of a reason described in the preceding paragraph, it is unpredictable whether the resulting instruction-length code is 0 or 1, or 0 or 2 if EXECUTE was used.

The operation is suppressed on all addressing and protection exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-25 on page 10-80.

**Resulting Condition Code:** The code is set as specified in the new PSW loaded.

**Program Exceptions:**
- Access (fetch and store, except key-controlled protection, linkage-stack entry)
- Addressing (authority-table entry, if SASN translation occurs)
- ASN translation (if PASN or SASN translation occurs)
- ASTE instance (if ASN-and-LX reuse is enabled and PASN or SASN translation occurs)
- Secondary authority (if SASN translation occurs)
- Space-switch event
- Special operation
- Specification
- Stack empty
- Stack operation
- Stack specification
- Stack type
- Subspace replacement (if PASN or SASN translation occurs)
- Trace
<table>
<thead>
<tr>
<th>Exception Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.</td>
</tr>
<tr>
<td>7. Special-operation exception due to DAT being off or the CPU being in secondary-space mode or home-space mode.</td>
</tr>
<tr>
<td>8.A Trace exceptions.</td>
</tr>
<tr>
<td>8.B.1 Access exceptions (fetch) for entry descriptor of the current linkage-stack entry.</td>
</tr>
<tr>
<td>8.B.2 Stack-type exception due to current entry not being a state entry or header entry.</td>
</tr>
<tr>
<td><strong>Note</strong>: Exceptions 8.B.3-8.B.7 can occur only if the current entry is a header entry.</td>
</tr>
<tr>
<td>8.B.3 Stack-operation exception due to unstack-suppression bit in the header entry being one.</td>
</tr>
<tr>
<td>8.B.4 Access exceptions (fetch) for second word of the header entry.</td>
</tr>
<tr>
<td>8.B.5 Stack-empty exception due to backward stack-entry validity bit in the header entry being zero.</td>
</tr>
<tr>
<td>8.B.6 Access exceptions (fetch) for entry descriptor of preceding entry, which is the entry designated by the backward stack-entry address in the current (header) entry.</td>
</tr>
<tr>
<td>8.B.7 Stack-specification exception due to preceding entry being a header entry.</td>
</tr>
<tr>
<td>8.B.8 Stack-type exception due to preceding entry not being a state entry.</td>
</tr>
<tr>
<td>8.B.9 Stack-operation exception due to unstack-suppression bit being one in the state entry.</td>
</tr>
<tr>
<td>8.B.10 Access exceptions (fetch) for the state entry, and access exceptions (store) for entry descriptor of the entry preceding the state entry.</td>
</tr>
<tr>
<td><strong>Note</strong>: Exceptions 8.B.11-8.B.15 and the event 9 can occur only if the state entry is a program-call state entry.</td>
</tr>
<tr>
<td>8.B.11 Special-operation exception due to the ASN-translation control, bit 44 of control register 14, being zero (if PASN or SASN translation occurs).</td>
</tr>
</tbody>
</table>

*Figure 10-25 (Part 1 of 2). Priority of Execution: PROGRAM RETURN*
8.B.12 ASN-translation exceptions (if PASN translation occurs).

8.B.13 ASTE-instance exception due to new PASTEIN not being equal to ASTEIN in ASN-second-table entry located by PASN translation (if ASN-and-LX reuse enabled).

Note: Subspace-replacement exceptions for replacement of bits in the PASCE, which are not shown in detail in this figure, can occur with any priority after 8.B.13 and before 9.

8.B.14 ASN-translation exceptions (if SASN translation occurs).

8.B.15 ASTE-instance exception due to new SASTEIN not being equal to ASTEIN in ASN-second-table entry located by SASN translation (if ASN-and-LX reuse enabled).

Note: Subspace-replacement exceptions for replacement of bits in the SASCE, which are not shown in detail in this figure, can occur with any priority after 8.B.15 and before 9.

8.B.16 Secondary-authority exception due to authority-table entry being outside table (if SASN translation occurs).

8.B.17 Addressing exception for access to authority-table entry (if SASN translation occurs).

8.B.18 Secondary-authority exception due to S bit in authority-table entry being zero (if SASN translation occurs).

9. Space-switch event (PR-ss only).

10. Specification exception due to any PSW error of the type that causes an immediate interruption.

Programming Note: Because PROGRAM CALL cannot be executed successfully in the secondary-space or home-space mode, PROGRAM RETURN is not intended to load a PSW specifying one of these translation modes. PROGRAM RETURN, unlike SET ADDRESS SPACE CONTROL and SET ADDRESS SPACE CONTROL FAST, does not recognize a space-switch event because of loading a PSW that specifies the home-space mode.

PROGRAM TRANSFER

```
PT R1,R2 [RRE]
```

```
'B228' // // // // R1 R2
0  16  24  28  31
```

Note: In this instruction definition, the name “PROGRAM TRANSFER (WITH INSTANCE)” refers to the PROGRAM TRANSFER instruction and the PROGRAM TRANSFER WITH INSTANCE instruction.

Bits 32-63 of general register R1 are used as the new values for the PSW-key mask, the PASN, and the SASN. Bits 32-63 or 0-63 of general register R2, depending on the current addressing
mode, are used as the new values for the problem-state bit, basic-addressing-mode bit, and instruction address in the current PSW. In the PROGRAM TRANSFER WITH INSTANCE operation, bits 0-31 of general register R₁ are an ASTEIN and are compared against the new value of the PASTEIN if the PASN is changed. In the PROGRAM TRANSFER operation, bits 0-31 of general register R₁ are ignored.

General registers R₁ and R₂ have the following format:

For PROGRAM TRANSFER

<table>
<thead>
<tr>
<th>R₁</th>
<th>PSW-Key Mask</th>
<th>ASN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 32 48 63</td>
<td></td>
</tr>
</tbody>
</table>

For PROGRAM TRANSFER WITH INSTANCE

<table>
<thead>
<tr>
<th>R₁</th>
<th>ASTEIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PSW-Key Mask</th>
<th>ASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 48 63</td>
<td></td>
</tr>
</tbody>
</table>

For PROGRAM TRANSFER (WITH INSTANCE)

In 24-Bit or 31-Bit Addressing Mode

<table>
<thead>
<tr>
<th>R₂</th>
<th>Instruction Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 32 63</td>
<td></td>
</tr>
</tbody>
</table>

In 64-Bit Addressing Mode

<table>
<thead>
<tr>
<th>R₂</th>
<th>Instruction Address</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 63</td>
<td></td>
</tr>
</tbody>
</table>

When the contents of bit positions 48-63 of general register R₁ are equal to the current PASN, the operation is called PROGRAM TRANSFER (WITH INSTANCE) to current primary (PT-cp or PTI-cp); when the fields are not equal, the operation is called PROGRAM TRANSFER (WITH INSTANCE) with space switching (PT-ss or PTI-ss).

The contents of general register R₂ are used to update the problem-state bit and the instruction address in the current PSW and, in the 24-bit or 31-bit addressing mode, also the basic-addressing-mode bit in the current PSW. Bit 63 of general register R₂ is placed in the problem-state bit position, PSW bit position 15, unless the operation would cause PSW bit 15 to change from one to zero (problem state to supervisor state). If such a change would occur, a privileged-operation exception is recognized.

In the 24-bit or 31-bit addressing mode, bit 32 of general register R₂ replaces the basic-addressing-mode bit, bit 32 of the current PSW, and bits 33-62 of the register, with one rightmost zero appended, replace bits 33-63 of the instruction address in the PSW, bits 97-127 of the PSW. In the 64-bit addressing mode, bits 0-62 of general register R₂, with one rightmost zero appended, replace the instruction address, and the basic-addressing-mode bit remains unchanged.

Bits 32-47 of general register R₁ are ANDed with the PSW-key mask, bits 32-47 of control register 3, and the result replaces the PSW-key mask. Bits 0-31 of general register R₁ are ignored.

In any of the PT-cp, PTI-cp, PT-ss, and PTI-ss operations, the ASN specified by bits 48-63 of general register R₁ replaces the SASN in control register 3, and the SASCE in control register 7 is replaced by the final contents of control register 1.

In the PROGRAM TRANSFER operation, if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the SASTEIN in control register 3 is replaced by the final value of the PASTEIN in control register 4. This replacement occurs in the PROGRAM TRANSFER WITH INSTANCE operation regardless of the value of the ASN-and-LX-reuse control.

PROGRAM TRANSFER (WITH INSTANCE) to Current Primary (PT-cp or PTI-cp)

The PT-cp operation is depicted in part 1 of Figure 10-27 on page 10-87. The PTI-cp operation is depicted in part 1 of Figure 10-28 on page 10-89. The PT-cp or PTI-cp operation is completed when the common portion of the PROGRAM TRANSFER (WITH INSTANCE) operation, described above, is completed. The PASTEIN, authorization index, PASN, primary ASCE, and contents of control register 5 (primary-ASN-second-table-entry origin) are not changed by PT-cp or PTI-cp. In this case in the PTI-cp operation, there is not a test of whether the
current PASTEIN equals the ASTEIN specified in bit positions 0-31 of general register R1; the ASTEIN is ignored.

**PROGRAM TRANSFER (WITH INSTANCE) with Space Switching (PT-ss or PTI-ss)**

If the ASN in bit positions 48-63 of general register R1 is not equal to the current PASN, a PROGRAM TRANSFER (WITH INSTANCE) with space switching (PT-ss or PTI-ss) operation is specified, and the ASN is translated by means of a two-level table lookup.

The PT-ss operation is depicted in parts 1 and 2 of Figure 10-27 on page 10-87. The PTI-ss operation is depicted in parts 1 and 2 of Figure 10-28 on page 10-89. The PT-ss or PTI-ss operation is completed as follows.

In PT-ss or PTI-ss, the contents of bit positions 48-63 of general register R1 are used as an ASN, which is translated by means of a two-level table lookup.

Bits 48-57 of general register R1 are a 10-bit AFX that is used to select an entry from the ASN first table. Bits 58-63 are a six-bit ASX that is used to select an entry from the ASN second table. The ASN table-lookup process is described in "ASN Translation" on page 3-24. The exceptions associated with ASN translation are collectively called “ASN-translation exceptions.” These exceptions and their priority are described in Chapter 6, "Interruptions.”

In PT-ss if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the reusable-ASN bit, bit 63, in the located ASN-second-table entry (ASTE) must be zero; otherwise, a special-operation exception is recognized. In PTI-ss, regardless of the ASN-and-LX-reuse control, the controlled-ASN bit, bit 62, in the ASTE must be zero if the CPU is in the problem state at the beginning of the operation; otherwise, a special-operation exception is recognized. Also in PTI-ss, and regardless of the ASN-and-LX-reuse control and the reusable-ASN bit, the ASTEIN in bit positions 0-31 of general register R1 must equal the ASTEIN in bit positions 352-383 of the ASTE; otherwise, an ASTE-instance exception is recognized.

The authority-table origin from the ASN-second-table entry (ASTE) is used as the base for a third table lookup. The current authorization index, bits 32-47 of control register 4, is used, after it has been checked against the authority-table length, as the index to locate the entry in the authority table. The authority-table lookup is described in "ASN Authorization" on page 3-30.

The PT-ss or PTI-ss operation is completed by placing bits 64-127 of the ASTE in control register 1 as the new PASCE and in control register 7 as the new SASCE. The contents of bit positions 32-47 of the ASTE replace the authorization index in bit positions 32-47 of control register 4. Bits 33-57 of the ASTE address are placed in bit positions 33-57 of control register 5 as the new primary-ASTE origin, and zeros are placed in bit positions 32 and 58-63. Bits 0-31 of this register remain unchanged. The ASN, bits 48-63 of general register R1, replaces the SASN and PASN in bit positions 48-63 of control registers 3 and 4.

In PT-ss if ASN-and-LX reuse is enabled, and in PTI-ss regardless of that enablement, the ASTEIN in the ASTE replaces the SASTEIN and PASTEIN in bit positions 0-31 of control registers 3 and 4.

The description in this paragraph applies to use of the subspace-group facility. After the new PASCE has been placed in control register 1 and the new primary-ASTE origin has been placed in control register 5, if (1) the subspace-group-control bit, bit 54, in the PASCE is one, (2) the dispatchable unit is subspace active, and (3) the primary-ASTE origin designates the ASTE for the base space of the dispatchable unit, then bits 0-55 and 58-63 of the PASCE in control register 1 are replaced by the same bits of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. This replacement occurs before a replacement of the SASCE in control register 7 by the PASCE. Further details are in "Subspace Replacement Operations" on page 5-68.

**PROGRAM TRANSFER (WITH INSTANCE) Serialization**

For any of the PT-cp, PTI-cp, PT-ss, and PTI-ss operations, a serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. However, it is unpredictable whether or not a store into a trace-table entry from which a
subsequent instruction is fetched will be observed by the CPU that performed the store.

**Special Conditions**

The instruction can be executed only when the CPU is in the primary-space mode and the subsystem-linkage control, bit 0 of the linkage-table designation, or linkage-first-table designation in the current primary ASN-second-table entry, is one. If the CPU is in the real mode, secondary-space mode, access-register mode, or home-space mode, or if the subsystem-linkage control is zero, a special-operation exception is recognized.

In PT-ss when ASN-and-LX reuse is enabled, the reusable-ASN bit in the ASN-second-table entry (ASTE) must be zero; otherwise, a special-operation exception is recognized. In PTI-ss, regardless of the ASN-and-LX-reuse control, a special-operation exception is recognized if the controlled-ASN bit in the ASTE is one and the CPU is in the problem state at the beginning of the operation, and an ASTE-instance exception is recognized if the ASTEIN in general register $R_1$ is not equal to the ASTEIN in the ASTE.

Bit 63 of general register $R_2$ is placed in the problem-state bit position, PSW bit position 15, unless the operation would cause PSW bit 15 to change from one to zero (problem state to supervisor state). If such a change would occur, a privileged-operation exception is recognized.

In the 24-bit or 31-bit addressing mode, the instruction is completed only if bits 32-39 of general register $R_2$ specify a valid combination of PSW bits 32 and 97-103. If bit 32 of general register $R_2$ is zero and bits 33-39 are not all zeros, a specification exception is recognized.

In addition to the above requirements, when a PT-ss or PTI-ss instruction is specified, the ASN-translation control, bit 44 of control register 14, must be one; otherwise, a special-operation exception is recognized.

When, for PT-ss or PTI-ss, the primary space-switch-event-control bit, bit 57 of register 1, is one either before or after the execution of the instruction, a space-switch-event program interruption occurs after the operation is completed. A space-switch-event program interruption also occurs after the completion of a PT-ss or PTI-ss operation if a PER event is reported.

The operation is suppressed on all addressing exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-26 on page 10-85.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Addressing (linkage-table or linkage-first-table designation in primary ASN-second-table entry; authority-table entry, PT-ss and PTI-ss only)
- ASN translation (PT-ss and PTI-ss only)
- ASTE instance (PTI-ss only)
- Operation (if ASN-and-LX-reuse facility is not installed, PTI only)
- Primary authority (PT-ss and PTI-ss only)
- Privileged operation (attempt to set the supervisor state when in the problem state)
- Space-switch event (PT-ss and PTI-ss only)
- Special operation
- Specification
- Subspace replacement (PT-ss and PTI-ss only)
- Trace
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B.1 Operation exception (PTI only, if the ASN-and-LX-reuse facility is not installed).

7.B.2 Special-operation exception due to DAT being off or the CPU being in secondary-space mode, access-register mode, or home-space mode.

8.A Trace exceptions.

8.B.1 Addressing exception for access to linkage-table designation or linkage-first-table designation in primary ASN-second-table entry.

8.B.2 Special-operation exception due to subsystem-linkage control in linkage-table designation or linkage-first-table designation being zero.

8.B.3 Privileged-operation exception due to attempt to set the supervisor state when in the problem state.

8.B.4 Specification exception due to invalid combination (bit 32 is zero and bits 33-39 not zeros) in general register R2 in 24-bit or 31-bit addressing mode.

8.B.5 Special-operation exception due to the ASN-translation control, bit 44 of control register 14, being zero (PT-ss or PTI-ss only).

8.B.6 ASN-translation exceptions (PT-ss and PTI-ss only).

8.B.7 Special-operation exception due to ASN-and-LX reuse enabled and reusable-ASN bit in ASN-second-table entry being one (PT-ss only).

8.B.8 Special-operation exception due to controlled-ASN bit in ASN-second-table entry being one and CPU being in problem state at the beginning of the operation (PTI-ss only).

8.B.9 ASTE-instance exception due to ASTEIN in general register R1 not being equal to ASTEIN in ASN-second-table entry (PTI-ss only).

Note: Subspace-replacement exceptions, which are not shown in detail in this figure, can occur with any priority after 8.B.9 and before 9.
8.B.10 Primary-authority exception due to authority-table entry being outside table (PT-ss and PTI-ss only).

8.B.11 Addressing exception for access to authority-table entry (PT-ss and PTI-ss only).

8.B.12 Primary-authority exception due to P bit in authority-table entry being zero (PT-ss and PTI-ss only).

9. Space-switch event (PT-ss and PTI-ss only).

---

Figure 10-26 (Part 2 of 2). Priority of Execution: PROGRAM TRANSFER (WITH INSTANCE)

**Programming Notes:**

1. The operation of PROGRAM TRANSFER (PT) is such that it may be used to restore the CPU to the state saved by a previous basic PROGRAM CALL operation. This restoration is accomplished by issuing PT 3,14. Though general registers 3 and 14 are not restored to their original values, the PASN, PSW-key mask, problem-state bit, and instruction address are restored, and the authorization index, PASCE, and primary-ASN-second-table-entry origin are made consistent with the restored PASN. In the 24-bit or 31-bit addressing mode, the basic-addressing-mode bit also is restored. If ASN-and-LX reuse is enabled, the PASTEIN also is made consistent with the restored PASN. Note that the SASN is not saved by PROGRAM CALL or restored by PROGRAM TRANSFER; PROGRAM TRANSFER sets the SASN equal to the restored PASN. PROGRAM TRANSFER WITH INSTANCE is the same as PROGRAM TRANSFER except that the PASTEIN is made consistent regardless of ASN-and-LX-reuse enablement.

2. With proper authority, and while being executed in a common area, PROGRAM TRANSFER (WITH INSTANCE) may be used to change the primary address space to any desired space. The secondary address space is also changed to be the same as the new primary address space.

3. Unlike the RR-format branch instructions, a value of zero in the R₁₁ field for PROGRAM TRANSFER (WITH INSTANCE) designates general register 0, and branching occurs.

4. A program given control by a basic PROGRAM CALL operation can use EXTRACT SECONDARY ASN AND INSTANCE to obtain the ASTEIN to be used by PROGRAM TRANSFER WITH INSTANCE to return to the calling program or by SET SECONDARY ASN WITH INSTANCE to restore its secondary address space after a change of that space. This EXTRACT SECONDARY ASN AND INSTANCE instruction should be executed while the original secondary space remains continuously the secondary space; otherwise, depending on actions by the control program, EXTRACT SECONDARY ASN AND INSTANCE may return an ASTEIN that allows return to or use of a conceptually incorrect secondary space for which the ASTEIN has been changed.
PT-cp and PT-ss

Figure 10-27 (Part 1 of 2). Execution of PROGRAM TRANSFER
Figure 10-27 (Part 2 of 2). Execution of PROGRAM TRANSFER
PTI-cp and PTI-ss

Program Transfer with Instance Instruction

'B99E' /// R1 R2

In 24-Bit or 31-Bit Mode

R2 32-63 A IA 33-62 P

With 33 Zeros on Left

In 64-Bit Mode

R2 IA 0-62 P

PTI-cp Instruction Complete

PTI-ss (See Part 2)

Figure 10-28 (Part 1 of 2). Execution of PROGRAM TRANSFER WITH INSTANCE
Figure 10-28 (Part 2 of 2). Execution of PROGRAM TRANSFER WITH INSTANCE
The ART-lookaside buffer (ALB) of this CPU is cleared of entries. No change is made to the contents of addressable storage or registers.

The ALB appears cleared of its original contents beginning with the execution of the next sequential instruction. The operation is not signaled to any other CPU.

A serialization function is performed.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Privileged operation

The translation-lookaside buffer (TLB) of this CPU is cleared of entries. No change is made to the contents of addressable storage or registers.

The TLB appears cleared of its original contents beginning with the fetching of the next sequential instruction. The operation is not signaled to any other CPU.

A serialization function is performed.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Privileged operation

The reference bit in the storage key for the 4K-byte block that is addressed by the contents of general register $R_2$ is set to zero. The contents of general register $R_1$ are ignored.

In the 24-bit addressing mode, bits 40-51 of general register $R_2$ designate a 4K-byte block in real storage, and bits 0-39 and 52-63 of the register are ignored. In the 31-bit addressing mode, bits 33-51 of general register $R_2$ designate a 4K-byte block in real storage, and bits 0-32 and 52-63 of the register are ignored. In the 64-bit addressing mode, bits 0-51 of general register $R_2$ designate a 4K-byte block in real storage, and bits 52-63 of the register are ignored.

Because it is a real address, the address designating the storage block is not subject to dynamic address translation. The reference to the storage key is not subject to a protection exception.

The remaining bits of the storage key, including the change bit, are not affected.

The condition code is set to reflect the state of the reference and change bits before the reference bit is set to zero.

**Resulting Condition Code:**
0 Reference bit zero; change bit zero
1 Reference bit zero; change bit one
2 Reference bit one; change bit zero
3 Reference bit one; change bit one

**Program Exceptions:**
- Addressing (address specified by general register $R_2$)
- Privileged operation
RESUME PROGRAM

RP Dz(Bz) [S]

<table>
<thead>
<tr>
<th>'B277'</th>
<th>Bz</th>
<th>Dz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Certain contents of the current PSW and of access register and general register Bz are replaced from three or four corresponding fields in the second operand. The size of the PSW field in the second operand, the size or number of general-register fields in the second operand, and the offsets of the fields in the second operand are specified in a parameter list that immediately follows the instruction in the instruction address space.

The instruction address space is the address space from which instructions are fetched. It is composed of real addresses if DAT is off.

The first 64 bits of the parameter list have the following format:

```
0000000000000000 | P | R | D | Offset of PSW Fld
0 | 13 | 16 | 31
```

Offset of AR Fld Offset of GR Fld 1

32 48 63

When bits 14 (R) and 15 (D) of the parameter list are both one, the list is an additional 16 bits in length, as follows:

```
Offset of GR Fld 2
64 79
```

Bit 13 of the parameter list (P) specifies the size of the PSW field in the second operand. The field is eight bytes if bit 13 is zero or 16 bytes if bit 13 is one.

Bits 14 and 15 of the parameter list (R and D) provide specifications about one or two general-register fields in the second operand, as follows:

- When bit 14 is zero, then bit 15 is ignored, the general-register field 1 in the second operand is four bytes, from which bits 32-63 of general register Bz will be replaced, there is not a general-register field 2 in the second operand, and bits 0-31 of general register Bz will remain unchanged.

- When bit 14 is one and bit 15 is zero, then the general-register field 1 is eight bytes, from which bits 0-63 of general register Bz will be replaced, and there is not a general-register field 2.

- When bits 14 and 15 are both one, then the general-register fields 1 and 2 are both four bytes, bits 32-63 of general register Bz will be replaced from field 1, and bits 0-31 of the register will be replaced from field 2. (The letter “D” stands for disjoint.)

Bits 16-31 of the parameter list are an unsigned binary integer that is the offset in bytes from the beginning of the second operand to a field that has the format of an eight-byte or 16-byte PSW, depending on bit 13, and from which fields in the current PSW will be replaced. Bits 32-47 similarly are an offset to a four-byte field from which the contents of access register Bz will be replaced. Bits 48-63 similarly are an offset to a four-byte or eight-byte field, depending on bits 14 and 15, from which bits 32-63 or 0-63, respectively, of general register Bz will be replaced. If bits 64-79 of the parameter list exist, they similarly are an offset to a four-byte field from which bits 0-31 of general register Bz will be replaced.

Bits 0-12 of the parameter list must be zeros; otherwise, a specification exception is recognized.

An eight-byte second-operand PSW field has the ESA/390 PSW format, as follows:

```
00000000 | I | E | T | X | Key | 1 | M | P | A | S | C | C | Mask | 00000000
0 | 5 | 8 | 12 | 16 | 18 | 20 | 24 | 31
```

A 16-byte second-operand PSW field has the z/Architecture PSW format, as follows:
Fields in the current PSW are replaced from the corresponding fields in the PSW field in the second operand. The PSW fields that are replaced are as follows:

<table>
<thead>
<tr>
<th>PSW Bits</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 and 17</td>
<td>Address-space control (AS)</td>
</tr>
<tr>
<td>18 and 19</td>
<td>Condition code (CC)</td>
</tr>
<tr>
<td>20-23</td>
<td>Program mask</td>
</tr>
<tr>
<td>31</td>
<td>Extended addressing mode (EA)</td>
</tr>
<tr>
<td>32</td>
<td>Basic addressing mode (BA)</td>
</tr>
<tr>
<td>64-127</td>
<td>Instruction address</td>
</tr>
</tbody>
</table>

The remaining fields in the PSW field in the second operand are ignored. Specifically, there is no test for whether bit 12 is one in an eight-byte PSW or zero in a 16-byte PSW. There is also no test for whether bit 31 is zero in an eight-byte PSW.

Unassigned fields in the PSW may be assigned in the future and may then be among those restored by RESUME PROGRAM. Therefore, these fields in the PSW field in the second operand should contain zeros; otherwise, the program may not operate compatibly in the future.

When PSW bits 64-127 are replaced from an eight-byte PSW field in the second operand, they are replaced with bits 33-63 of the field, with 33 zeros appended on the left.

The fields in the second operand are fetched before the contents of access register B₂ and general register B₂ are changed.

When RESUME PROGRAM is the target of an EXECUTE instruction, the parameter list immediately follows the RESUME PROGRAM instruction, not the EXECUTE instruction.

The references to the parameter list are storage-operand fetches, not instruction fetches.

**Special Conditions**

The instruction is completed only if the bits 31, 32, and 64-127 that are to be placed in the current PSW are valid for placement in the PSW. If bits 31 and 32 are both zero and bits 64-103 are not all zeros, if bits 31 and 32 are zero and one, respectively, and bits 64-96 are not all zeros, if bits 31 and 32 are one and zero, respectively, or if bit 127 is one, a specification exception is recognized.

The CPU must be in the supervisor state when the operation is to set the home-space mode; otherwise, a privileged-operation exception is recognized. When DAT is off, the values of bits 16 and 17 of the PSW field in the second operand are not tested.

When the CPU is in the home-space mode either before or after the operation, but not both before and after the operation, a space-switch-event program interruption occurs after the operation is completed if any of the following is true: (1) the primary space-switch-event control, bit 57 of the primary address-space-control element (ASCE) in control register 1, is one; (2) the home space-switch-event control, bit 57 of the home ASCE in control register 13, is one; or (3) a PER event is to be indicated.

The operation is suppressed on all addressing and protection exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-29 on page 10-94.

**Resulting Condition Code:** The code is set as specified by the new condition code loaded.

**Program Exceptions:**
- Access (fetch, parameter list and operand 2)
- Privileged operation (attempt to set the home-space mode when in the problem state)
- Space-switch event
Programming Notes:

1. As described in "Instruction Fetching" on page 5-93, the bytes of an instruction may be fetched piecemeal, and the instruction may be fetched multiple times for a single execution. Therefore, the results are unpredictable when instructions are fetched for execution from storage that is being changed by another CPU or a channel program. This warning is particularly applicable when RESUME PROGRAM is the target of EXECUTE since the EXECUTE instruction may be refetched in order to generate, from its B, X, and D fields, the address of the parameter list used by RESUME PROGRAM. If EXECUTE is refetched, there is not necessarily a test for whether storage still contains either the EXECUTE instruction or the RESUME PROGRAM instruction.

2. The storage-operand references for RESUME PROGRAM may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98.)

| 1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case. |
| 8.A Trace exceptions. |
| 8.B.1 Access exceptions for bits 0-63 of parameter list. |
| 8.B.2 Specification exception due to bits 0-12 of parameter list not being all zeros. |
| 8.B.3 Access exceptions for bits 64-79 of parameter list, if these bits exist. |
| 8.B.5 Privileged-operation exception due to attempt to set the home-space mode when in the problem state. |
| 8.B.6 Specification exception due to invalid values in bit positions 31, 32, and 64-127 of PSW in second operand. |
| 9. Space-switch event. |

Figure 10-29. Priority of Execution: RESUME PROGRAM
**SET ADDRESS SPACE CONTROL**

**SAC** \( D_z(B_z) \) \[S\]

<table>
<thead>
<tr>
<th>'B219'</th>
<th>B_z</th>
<th>D_z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

**SET ADDRESS SPACE CONTROL FAST**

**SACF** \( D_z(B_z) \) \[S\]

<table>
<thead>
<tr>
<th>'B279'</th>
<th>B_z</th>
<th>D_z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Bits 52-55 of the second-operand address are used as a code to set the address-space-control bits in the PSW. The second-operand address is not used to address data; instead, bits 52-55 form the code. Bits 0-51 and 56-63 of the second-operand address are ignored. Bits 52 and 53 of the second-operand address must be zeros; otherwise, a specification exception is recognized.

The following figure summarizes the operation of SET ADDRESS SPACE CONTROL and SET ADDRESS SPACE CONTROL FAST:

```
Second-Operand Address

          Code
----------
0          52 56 63

Result in PSW Bits 16 and 17

<table>
<thead>
<tr>
<th>Code</th>
<th>Name of Mode</th>
<th>PSM Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Primary space</td>
<td>00</td>
</tr>
<tr>
<td>0001</td>
<td>Secondary space</td>
<td>10</td>
</tr>
<tr>
<td>0010</td>
<td>Access register</td>
<td>01</td>
</tr>
<tr>
<td>0011</td>
<td>Home space</td>
<td>11</td>
</tr>
<tr>
<td>All others</td>
<td>Invalid</td>
<td>Unchanged</td>
</tr>
</tbody>
</table>
```

The CPU must be in the supervisor state when the operation is to set the home-space mode; otherwise, a privileged-operation exception is recognized.

For SET ADDRESS SPACE CONTROL, a serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. This function is not performed for SET ADDRESS SPACE CONTROL FAST.

**Special Conditions**

For SET ADDRESS SPACE CONTROL, the operation is performed only when the secondary-space control, bit 37 of control register 0, is one and DAT is on. When either the secondary-space control is zero or DAT is off, a special-operation exception is recognized. The same rules apply also to SET ADDRESS SPACE CONTROL FAST, except that whether the secondary-space control is tested is unpredictable.

When the CPU is in the home-space mode either before or after the operation, but not both before and after the operation, a space-switch-event program interruption occurs after the operation is completed if any of the following is true: (1) the primary space-switch-event control, bit 57 of the primary address-space-control element (ASCE) in control register 1, is one; (2) the home space-switch-event control, bit 57 of the home ASCE in control register 13, is one; or (3) a PER event is to be indicated.

The priority of recognition of program exceptions for the instructions is shown in Figure 10-30 on page 10-96.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Privileged operation (attempt to set the home-space mode in the problem state)
- Space-switch event
- Special operation
- Specification
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Special-operation exception due to DAT being off.

7.C Special-operation exception due to the secondary-space control, bit 37 of control register /zerodot, being zero. May be omitted for SET ADDRESS SPACE CONTROL FAST.

8. Privileged-operation exception due to attempt to set home-space mode when in problem state.

9. Specification exception due to non-zero value in bit positions 52 and 53 of second-operand address.

10. Space-switch event.

The current value of the TOD clock is replaced by the contents of the doubleword designated by the second-operand address, and the clock enters the stopped state.

The doubleword operand replaces the contents of the clock, as determined by the resolution of the clock. Only those bits of the operand are set in the clock that correspond to the bit positions which are updated by the clock; the contents of the remaining rightmost bit positions of the operand are ignored and are not preserved in the clock. In some models, starting at or to the right of bit position 52, the rightmost bits of the second operand are ignored, and the corresponding positions of the clock which are implemented are set to zeros. Zeros are also placed in positions to the right of bit position 63 of the clock.

After the clock value is set, the clock enters the stopped state. The clock leaves the stopped state to enter the set state and resume incrementing under control of the TOD-clock-sync control, bit 34 of control register 0, of the CPU which most recently caused the clock to enter the stopped state. When the bit is zero, the clock enters the set state at the completion of the instruction. When the bit is one, the clock remains in the stopped state until the bit is set to zero or until another CPU executes a SET CLOCK instruction.
affecting the clock. If an external time reference (ETR) is installed, a signal from the ETR may be used to set the set state from the stopped state.

The value of the clock is changed and the clock is placed in the stopped state only if the manual TOD-clock control of any CPU in the configuration is set to the enable-set position or the TOD-clock-control-override control, bit 42 of control register 14, is one. If the TOD-clock control of all CPUs is set to the secure position and the TOD-clock-control-override control is zero, the value and state of the clock are not changed. Whether the clock is set or remains unchanged is distinguished by condition codes 0 and 1, respectively.

When the clock is not operational, the value and state of the clock are not changed, regardless of the settings of the TOD-clock control and the TOD-clock-control-override control, and condition code 3 is set.

Special Conditions

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

Resulting Condition Code:

0   Clock value set
1   Clock value secure
2   --
3   Clock in not-operational state

Program Exceptions:

- Access (fetch, operand 2)
- Privileged operation
- Specification

SET CLOCK PROGRAMMABLE FIELD

SCKPF [E]

\[
\begin{array}{c}
0 \\
15 \\
\end{array}
\]

Bits 48-63 of general register 0 are placed in bit positions 16-31 of the TOD programmable register. Zeros are placed in bit positions 0-15 of the TOD programmable register.

Special Conditions

Bits 32-47 of general register 0 must be zeros; otherwise, a specification exception is recognized. Bits 0-31 of general register 0 are ignored.

Condition Code: The code remains unchanged.

Program Exceptions:

- Privileged operation
- Specification

Programming Note: The values in the TOD programmable registers of a configuration should be the same and should be unique within a multiple-configuration system.
SET CPU TIMER

SPT  Dz(Bz)  [S]

<table>
<thead>
<tr>
<th>'B208'</th>
<th>Bz</th>
<th>Dz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The current value of the CPU timer is replaced by the contents of the doubleword designated by the second-operand address.

Only those bits of the operand are set in the CPU timer that correspond to the bit positions to be updated; the contents of the remaining rightmost bit positions of the operand are ignored and are not preserved in the CPU timer.

Special Conditions

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

The operation is suppressed on all addressing and protection exceptions.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2)
- Privileged operation
- Specification

SET PREFIX

SPX  Dz(Bz)  [S]

<table>
<thead>
<tr>
<th>'B210'</th>
<th>Bz</th>
<th>Dz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The contents of bit positions 33-50 of the prefix register are replaced by the contents of bit positions 1-18 of the word at the location designated by the second-operand address. The ART-lookaside buffer (ALB) and translation-lookaside buffer (TLB) of this CPU are cleared of entries.

After the second operand is fetched, the value is tested for validity before it is used to replace the contents of the prefix register. Bits 1-18 of the operand with 13 zeros appended on the right and 33 zeros appended on the left are used as an absolute address of the 8K-byte new prefix area in storage. This address is treated as a 64-bit address regardless of the addressing mode specified by the current PSW. The two 4K-byte blocks within the new prefix area are accessed; if either is not available in the configuration, an addressing exception is recognized, and the operation is suppressed. The accesses to the blocks are not subject to protection; however, the accesses may cause the reference bits for the blocks to be set to ones.

If the operation is completed, the new prefix is used for any interruptions following the execution of the instruction and for the execution of subsequent instructions. The contents of bit positions 0 and 19-31 of the second operand are ignored.

The ART-lookaside buffer (ALB) and translation-lookaside buffer (TLB) are cleared of entries. The ALB and TLB appear cleared of their original contents, beginning with the fetching of the next sequential instruction.

A serialization function is performed before or after the second operand is fetched and again after the operation is completed.

Special Conditions

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

The operation is suppressed on all addressing and protection exceptions.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2)
- Addressing (new prefix area)
- Privileged operation
- Specification
SET PSW KEY FROM ADDRESS

SPKA  D2(B2)  [S]

The four-bit PSW key, bits 8-11 of the current PSW, is replaced by bits 56-59 of the second-operand address.

The second-operand address is not used to address data; instead, bits 56-59 of the address form the new PSW key. Bits 0-55 and 60-63 of the second-operand address are ignored.

Special Conditions

In the problem state, the execution of the instruction is subject to control by the PSW-key mask in control register 3. When the bit in the PSW-key mask corresponding to the PSW-key value to be set is one, the instruction is executed successfully. When the selected bit in the PSW-key mask is zero, a privileged-operation exception is recognized. In the supervisor state, any value for the PSW key is valid.

Condition Code: The code remains unchanged.

Program Exceptions:

- Privileged operation (selected PSW-key-mask bit is zero in the problem state)

Programming Notes:

1. The format of SET PSW KEY FROM ADDRESS permits the program to set the PSW key either from the general register designated by the B2 field or from the D2 field in the instruction itself.

2. When one program requests another program to access a location designated by the requesting program, SET PSW KEY FROM ADDRESS can be used by the called program to verify that the requesting program is authorized to make this access, provided the storage location of the called program is not protected against fetching. The called program can perform the verification by replacing the PSW key with the requesting-program PSW key before making the access and subsequently restoring the called-program PSW key to its original value. Caution must be exercised, however, in handling any resulting protection exceptions since such exceptions may cause the operation to be terminated. See TEST PROTECTION and the associated programming notes for an alternative approach to the testing of addresses passed by a calling program.

SET SECONDARY ASN

SSAR  R1  [RRE]

The ASN specified in bit positions 48-63 of general register R1 replaces the secondary ASN in control register 3, and the address-space-control element corresponding to that ASN replaces the SASCE in control register 7.

The contents of bit positions 48-63 of general register R1 are called the new ASN. For SET SECONDARY ASN WITH INSTANCE, bits 0-31 of the register are an ASTEIN and are compared against the new SASTEIN if the SASTEIN is not set equal to the PASTEIN, and bits 32-47 of the register are ignored. For SET SECONDARY ASN, bits 0-47 of the register are ignored.

Note: In this instruction definition, the name “SET SECONDARY ASN (WITH INSTANCE)” refers to the SET SECONDARY ASN instruction and the SET SECONDARY ASN WITH INSTANCE instruction.

When the new ASN is equal to the current PASN, the operation is called SET SECONDARY ASN (WITH INSTANCE) to current primary (SSAR-cp or SSAIR-cp); when the fields are not equal, the operation is called SET SECONDARY ASN (WITH INSTANCE) with space switching (SSAR-ss or SSAIR-ss).
The SSAR-cp and SSAR-ss operations are depicted in Figure 10-32 on page 10-103. The SSAIR-cp and SSAIR-ss operations are depicted in Figure 10-33 on page 10-104.

SET SECONDARY ASN (WITH INSTANCE) to Current Primary (SSAR-cp or SSAIR-cp)

In the SSAR-cp or SSAIR-cp operation, the new ASN is equal to the PASN. The new ASN replaces the SASN, bits 48-63 of control register 3; the PASCE in control register 1 replaces the SASCE in control register 7.

In SSAR-cp, if the ASN-and-LX-reuse facility is installed and is enabled by a one value of the ASN-and-LX-reuse control, bit 44 of control register 0, the SASTEIN in bit positions 0-31 of control register 3 is replaced by the PASTEIN in bit positions 0-31 of control register 4. This replacement occurs in SSAIR-cp regardless of the value of the ASN-and-LX-reuse control. In SSAIR-cp, there is not a test of whether the current PASTEIN equals the ASTEIN in bit positions 0-31 of general register R; the ASTEIN is ignored. The operation is completed.

SET SECONDARY ASN (WITH INSTANCE) with Space Switching (SSAR-ss or SSAIR-ss)

In the SSAR-ss or SSAIR-ss operation, the new ASN is not equal to the PASN, and the new ASN is translated by means of a two-level table lookup. Bits 0-9 of the new ASN (bits 48-57 of the register) are a 10-bit AFX which is used to select an entry from the ASN first table. Bits 10-15 of the new ASN (bits 58-63 of the register) are a six-bit ASX which is used to select an entry from the ASN second table. The two-level lookup is described in "ASN Translation" on page 3-24. The exceptions associated with ASN translation are collectively called "ASN-translation exceptions." These exceptions and their priority are described in Chapter 6, "Interruptions."

In SSAR-ss, if the ASN-and-LX-reuse facility is installed and is enabled by the ASN-and-LX-reuse control in control register 0, the reusable-ASN bit, bit 63, in the located ASN-second-table entry (ASTE) must be zero; otherwise, a special-operation exception is recognized. In SSAIR-ss, regardless of the ASN-and-LX-reuse control, the controlled-ASN bit, bit 62, in the ASTE must be zero in the problem state; otherwise, a special-operation exception is recognized. Also in SSAIR-ss, and regardless of the ASN-and-LX-reuse control and the reusable-ASN bit, the ASTEIN in bit positions 0-31 of general register R must equal the ASTEIN in bit positions 352-383 of the ASTE; otherwise, an ASTE-instance exception is recognized.

The ASN-second-table entry (ASTE) obtained as a result of the second lookup contains the address-space-control element and the authority-table origin and length associated with the ASN.

The authority-table origin from the ASTE is used as a base for a third table lookup. The current authorization index, bits 32-47 of control register 4, is used, after it has been checked against the authority-table length, as the index to locate the entry in the authority table. The authority-table lookup is described in "ASN Authorization" on page 3-30.

The new ASN, bits 48-63 of general register R, replaces the SASN, bits 48-63 of control register 3. The address-space-control element in the ASTE replaces the SASCE in control register 7. In SSAR-ss if ASN-and-LX reuse is enabled, and in SSAIR-ss regardless of that enablement, the ASTEIN in the ASTE replaces the SASTEIN in bit positions 0-31 of control register 3.

The description in this paragraph applies to use of the subspace-group facility. After the new SASCE has been placed in control register 7, if (1) the subspace-group-control bit, bit 54, in the SASCE is one, (2) the dispatchable unit is subspace active, and (3) the ASTE obtained by ASN translation is the ASTE for the base space of the dispatchable unit, then bits 0-55 and 58-63 of the SASCE are replaced by the same bits of the ASCE in the ASTE for the subspace in which the dispatchable unit last had control. Further details are in "Subspace-Replacement Operations" on page 5-68.

SET SECONDARY ASN (WITH INSTANCE) Serialization

For any of the SSAR-cp, SSAIR-cp, SSAR-ss, and SSAIR-ss operations, a serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. However, it is unpredict-
able whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.

**Special Conditions**

The operation is performed only when the ASN-translation control, bit 44 of control register 14, is one and DAT is on. When either the ASN-translation-control bit is zero or DAT is off, a special-operation exception is recognized.

In SSAR-ss when ASN-and-LX reuse is enabled, the reusable-ASN bit in the ASN-second-table entry (ASTE) must be zero; otherwise, a special-operation exception is recognized. In SSAIR-ss, regardless of the ASN-and-LX-reuse control, a special-operation exception is recognized if the controlled-ASN bit in the ASTE is one and the CPU is in the problem state, and an ASTE-instance exception is recognized if the ASTEIN in general register R is not equal to the ASTEIN in the ASTE.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-31 on page 10-102.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Addressing (authority-table entry, SSAR-ss and SSAIR-ss only)
- ASN translation (SSAR-ss and SSAIR-ss only)
- ASTE instance (SSAIR-ss only)
- Operation (if ASN-and-LX-reuse facility is not installed, SSAIR only)
- Secondary authority (SSAR-ss and SSAIR-ss only)
- Special operation
- Subspace replacement (SSAR-ss and SSAIR-ss only)
- Trace

**Programming Note:** A program given control by a basic PROGRAM CALL operation can use EXTRACT SECONDARY ASN AND INSTANCE to obtain the ASTEIN to be used by PROGRAM TRANSFER WITH INSTANCE to return to the calling program or by SET SECONDARY ASN WITH INSTANCE to restore its secondary address space after a change of that space. This EXTRACT SECONDARY ASN AND INSTANCE instruction should be executed while the original secondary space remains continuously the secondary space; otherwise, depending on actions by the control program, EXTRACT SECONDARY ASN AND INSTANCE may return an ASTEIN that allows return to or use of a conceptually incorrect secondary space for which the ASTEIN has been changed.
1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B.1 Operation exception (SSAIR only, if the ASN-and-LX-reuse facility is not installed).

7.B.2 Special-operation exception due to DAT being off, or the ASN-translation control, bit 44 of control register 14, being zero.

8.A Trace exceptions.

8.B.1 ASN-translation exceptions (SSAR-ss and SSAIR-ss only).

8.B.2 Special-operation exception due to ASN-and-LX reuse enabled and reusable-ASN bit in ASN-second-table entry being one (SSAR-ss only).

8.B.3 Special-operation exception due to controlled-ASN bit in ASN-second-table entry being one in the problem state (SSAIR-ss only).

8.B.4 ASTE-instance exception due to ASTEIN in general register R not being equal to ASTEIN in ASN-second-table entry (SSAIR-ss only).

   Note: Subspace-replacement exceptions, which are not shown in detail in this figure, can occur with any priority after 8.B.4.

8.B.5 Secondary-authority exception due to authority-table entry being outside table (SSAR-ss and SSAIR-ss only).

8.B.6 Addressing exception for access to authority-table entry (SSAR-ss and SSAIR-ss only).

8.B.7 Secondary-authority exception due to S bit in authority-table entry being zero (SSAR-ss and SSAIR-ss only).

Figure 10-31. Priority of Execution: SET SECONDARY ASN (WITH INSTANCE)
Figure 10-32. Execution of SET SECONDARY ASN
Figure 10-33. Execution of SET SECONDARY ASN WITH INSTANCE
The storage key for the 4K-byte block that is addressed by the contents of general register R₂ is replaced by bits from general register R₁.

In the 24-bit addressing mode, bits 40-51 of general register R₂ designate a 4K-byte block in real storage, and bits 0-39 and 52-63 of the register are ignored. In the 31-bit addressing mode, bits 33-51 of general register R₂ designate a 4K-byte block in real storage, and bits 0-32 and 52-63 of the register are ignored. In the 64-bit addressing mode, bits 0-51 of general register R₂ designate a 4K-byte block in real storage, and bits 52-63 of the register are ignored.

Because it is a real address, the address designating the storage block is not subject to dynamic address translation. The reference to the storage key is not subject to a protection exception.

The new seven-bit storage-key value is obtained from bit positions 56-62 of general register R₁. The contents of bit positions 0-55 and 63 of the register are ignored.

A serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed.

For any store access, by any CPU or channel program, completed to the designated 4K-byte block either before or after the execution of this instruction, the associated setting of the reference and change bits to one in the storage key for the block also is completed before or after, respectively, the execution of this instruction.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Addressing (address specified by general register R₂)
- Privileged operation

---

**SET SYSTEM MASK**

<table>
<thead>
<tr>
<th>SSM</th>
<th>D₂(B₂)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'80'</td>
<td>//</td>
<td>B₂</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

Bits 0-7 of the current PSW are replaced by the byte at the location designated by the second-operand address.

**Special Conditions**

When the SSM-suppression-control bit, bit 33 of control register 0, is one and the CPU is in the supervisor state, a special-operation exception is recognized.

The value to be loaded into the PSW is not checked for validity before loading. However, immediately after loading, a specification exception is recognized, and a program interruption occurs, if the contents of bit positions 0 and 2-4 of the PSW are not all zeros. In this case, the instruction is completed, and the instruction-length code is set to 2. The specification exception, which is listed as a program exception for this instruction, is described in "Early Exception Recognition" on page 6-9. This exception may be considered as caused by execution of this instruction or as occurring early in the process of preparing to execute the subsequent instruction.

The operation is suppressed on all addressing and protection exceptions.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2)
- Privileged operation
- Special operation
- Specification
An eight-bit order code and, if called for, a 32-bit parameter are transmitted to the CPU designated by the CPU address contained in the third operand. The result is indicated by the condition code and may be detailed by status assembled in bit positions 32-63 of the first-operand location.

The second-operand address is not used to address data; instead, bits 56-63 of the address contain the eight-bit order code. Bits 0-55 of the second-operand address are ignored. The order code specifies the function to be performed by the addressed CPU. The assignment and definition of order codes appear in "CPU Signaling and Response" on page 4-56.

The 16-bit binary number contained in bit positions 48-63 of general register R₃ forms the CPU address. Bits 0-47 of the register are ignored. When the specified order is the set-architecture order, the CPU address is ignored; all other CPUs in the configuration are considered to be addressed.

The general register containing the 32-bit parameter in bit positions 32-63 is R₁ or R₁+1, whichever is the odd-numbered register. It depends on the order code whether a parameter is provided and for what purpose it is used.

The operands just described have the following formats:

General register designated by R₁:

<table>
<thead>
<tr>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 32 63</td>
</tr>
</tbody>
</table>

General register designated by R₁ or R₁ + 1, whichever is the odd-numbered register:

<table>
<thead>
<tr>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 32 63</td>
</tr>
</tbody>
</table>

A serialization function is performed before the operation begins and again after the operation is completed.

When the order code is accepted and no nonzero status is returned, condition code 0 is set. When status information is generated by this CPU or returned by the addressed CPU, the status is placed in bit positions 32-63 of general register R₁, bits 0-31 of the register remain unchanged, and condition code 1 is set.

When the access path to the addressed CPU is busy, or the addressed CPU is operational but in a state where it cannot respond to the order code, condition code 2 is set.

When the addressed CPU is not operational (that is, it is not provided in the installation, it is not in the configuration, it is in any of certain customer-engineer test modes, or its power is off), condition code 3 is set.

**Resulting Condition Code:**

0  Order code accepted  
1  Status stored  
2  Busy  
3  Not operational

**Program Exceptions:**

- Privileged operation

**Programming Notes:**

1. A more detailed discussion of the condition-code settings for SIGNAL PROCESSOR is contained in "CPU Signaling and Response" on page 4-56.

2. To ensure that presently written programs will be executed properly when new facilities using additional bits are installed, only zeros should
appears in the unused bit positions of the second-operand address and in bit positions 32-47 of general register \( R_3 \).

3. Certain SIGNAL PROCESSOR orders are provided with the expectation that they will be used primarily in special circumstances. Such orders may be implemented with the aid of an auxiliary maintenance or service processor, and, thus, the execution time may take several seconds. Unless all of the functions provided by the order are required, combinations of other orders, in conjunction with appropriate programming support, can be expected to provide a specific function more rapidly. The emergency-signal, external-call, and sense orders are the only orders which are intended for frequent use. The following orders are intended for infrequent use, and performance therefore may be much slower than for frequently used orders: restart, set prefix, store status at address, start, stop, stop and store status, set architecture, and all the reset orders. An alternative to the set-prefix order, for faster performance when the receiving CPU is not already stopped, is the use of the emergency-signal or external-call order, followed by the execution of a SET PREFIX instruction on the addressed CPU. Clearing the TLB of entries is ordinarily accomplished more rapidly through the use of the emergency-signal or external-call order, followed by execution of the PURGE TLB instruction on the addressed CPU.

**STORE CLOCK COMPARATOR**

**STORE CONTROL**

<table>
<thead>
<tr>
<th>STCKC D2(B2)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B207'</td>
<td>B2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

The current value of the clock comparator is stored at the doubleword location designated by the second-operand address.

Zeros are provided for the rightmost bit positions of the clock comparator that are not compared with the TOD clock.

**Special Conditions**

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (store, operand 2)
- Privileged operation
- Specification

**STORE CONTROL**

<table>
<thead>
<tr>
<th>STCTL R1, R3, D2(B2) [RS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B6' R1 R3 B2 D2</td>
</tr>
<tr>
<td>0 8 12 16 20 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STCTG R1, R3, D2(B2) [RSY]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EB' R1 R3 B2 DL2 DH2 '25'</td>
</tr>
<tr>
<td>0 8 12 16 20 32 40 47</td>
</tr>
</tbody>
</table>

Bits of the set of control registers starting with control register \( R_1 \) and ending with control register \( R_3 \) are stored at the locations designated by the second-operand address.

For STORE CONTROL (STCTL), bits 32-63 of the control registers are stored in successive words beginning at the second-operand address, and bits 0-31 of the registers are ignored. For STORE CONTROL (STCTG), bits 0-63 of the control registers are stored in successive doublewords beginning at the second-operand address.

The storage area where the contents of the control registers are placed starts at the location designated by the second-operand address and continues through as many storage words, for STCTL, or doublewords, for STCTG, as the number of control registers specified. The contents of the control registers are stored in ascending order of their register numbers, starting with control register \( R_1 \) and continuing up to and including control register \( R_3 \), with control register 0 following control register 15. The contents of the control registers remain unchanged.
The displacement for STCTL is treated as a 12-bit unsigned binary integer. The displacement for STCTG is treated as a 20-bit signed binary integer.

**Special Conditions**

The second operand must be designated on a word boundary for STCTL or on a doubleword boundary for STCTG; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (store, operand 2)
- Privileged operation
- Specification

### STORE CPU ADDRESS

**STORE CPU ID**

STIDP Dz(Bz) [S]

<table>
<thead>
<tr>
<th>B20Z</th>
<th>Bz</th>
<th>Dz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Information identifying either (1) the CPU or (2) the configuration and the logical partition in which the program is being executed is stored at the doubleword location designated by the second-operand address.

The information stored has the following format:

<table>
<thead>
<tr>
<th>CPU Identification Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine-Type Number</th>
<th>F</th>
<th>0000000000000000000000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>48</td>
<td>63</td>
</tr>
</tbody>
</table>

Bit positions 8-31 contain the CPU identification number, consisting of six four-bit digits. Some or all of these digits are selected from the physical serial number stamped on the CPU.

The format bit (F) in bit position 48 specifies the format of the first two digits of the CPU identification number.

When the format bit is zero, the contents of the CPU-identification-number field, in conjunction with the machine-type number, permit unique identification of the CPU. When the format bit is one, the CPU-identification number identifies the system configuration as opposed to an individual CPU within the configuration, and it identifies the logical partition in which the program is being executed.

Bit positions 32-47 contain the machine-type number of the CPU. Bit positions 0-7 and 49-63 contain zeros.
Special Conditions

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (store, operand 2)
- Privileged operation
- Specification

Programming Notes:

1. The program should allow for the possibility that the CPU identification number may contain the digits A-F as well as the digits 0-9.

2. When the format bit in bit position 48 of the second operand is zero, the CPU identification number, in conjunction with the machine-type number, provides a unique CPU identification that can be used in associating results with an individual machine.

3. In versions of Enterprise Systems Architecture/390 Principles of Operation prior to SA22-7201-03, the machine-type-number field was called the model-number field. When the format bit is one, the CPU-identification number identifies the system configuration as opposed to an individual CPU within the configuration, and it identifies the requesting logical partition. The CPU identity can be obtained using the STORE CPU ADDRESS or the STORE SYSTEM INFORMATION instruction.

4. In ESA/390, bit positions 0-7 of the information stored contain a version code. When the version code is nonzero, it is usually indicative of the model number of the model and the number of CPUs contained in the model. In both ESA/390 and z/Architecture, the STORE SYSTEM INFORMATION instruction can be used to determine the model number and the number of CPUs in the model.

5. The CPU identification number has the hex format:
   - “Annbnm” in the basic mode, or
   - “LPnnnn” in the LPAR (logically-partitioned) mode, when the format bit is zero, or
   - “PPnnnn” in the LPAR mode, when the format bit is one.

Where:

- A is the CPU address of the CPU.
- L is a logical CPU address.
- P is a logical-partition identifier (UPID). The UPID is an eight-bit unsigned binary integer bound to a logical partition.
- n is a digit derived from the serial number of the CPU.

The terminology above that is not defined in this publication is defined in the appropriate System Library publications for the model.

6. The format bit is always stored as a zero in the basic mode.

7. Model z800 and z900 machines always store the format bit as a zero.

8. When STORE CPU ID is executed in a virtual machine, the VM operating system replaces bits 0-7 of the information stored with FF hex.

STORE CPU TIMER

```
STPT D2(B2) [S]

┌────────────────┬────┬────────────┐
│ ‘B209’         │ B2 │ D2         │
└────────────────┴────┴────────────┘
```

The current value of the CPU timer is stored at the doubleword location designated by the second-operand address.

Zeros are provided for the rightmost bit positions that are not updated by the CPU timer.

Special Conditions

The operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (store, operand 2)
- Privileged operation
- Specification
A list of bits providing information about facilities is stored in the word at real address 200. The bits have meanings as follows.

**Bit Meaning When Bit Is One**

0 The instructions marked “N3” in the instruction-summary figures in Chapters 7 and 10 are installed.

1 The z/Architecture architectural mode is installed.

2 The z/Architecture architectural mode is active. When this bit is zero, the ESA/390 architectural mode is active.

3 The DAT-enhancement facility is installed in the z/Architecture architectural mode. The DAT-enhancement facility includes the INVALIDATE DAT TABLE ENTRY (IDTE) and COMPARE AND SWAP AND PURGE (CSPG) instructions.

4 INVALIDATE DAT TABLE ENTRY (IDTE) performs the invalidation-and-clearing operation by selectively clearing combined region-and-segment-table entries when a segment-table entry or entries are invalidated. IDTE also performs the clearing-by-ASCE operation. Bit 3 is one if bit 4 is one.

5 INVALIDATE DAT TABLE ENTRY (IDTE) performs the invalidation-and-clearing operation by selectively clearing combined region-and-segment-table entries when a region-table entry or entries are invalidated. Bits 3 and 4 are ones if bit 5 is one.

6 The ASN-and-LX reuse facility is installed in the z/Architecture architectural mode.

16 The extended-translation facility 2 is installed.

17 The message-security assist is installed.

18 The long-displacement facility is installed in the z/Architecture architectural mode.

19 The long-displacement facility has high performance. Bit 18 is one if bit 19 is one.

20 The HFP-multiply-add/subtract facility is installed.

22 The extended-translation facility 3 is installed in the z/Architecture architectural mode.

A bit is set to one regardless of the current architectural mode if its meaning is true. A meaning applies to the current architectural mode unless it is said to apply to a specific architectural mode.

Bits 7-15, 21, and 23-31 are reserved for indication of new facilities. The bits are currently stored as zeros but may be stored as ones in the future.

The second-operand address is ignored but should be zero to permit possible future extensions.

Key-controlled and low-address protection do not apply.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Privileged operation

---

The contents of bit positions 33-50 of the prefix register are stored in bit positions 1-18 of the word location designated by the second-operand address, and zeros are stored in bit positions 0 and 19-31 of the word.

**Special Conditions**

The operand must be designated on a word boundary; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (store, operand 2)
- Privileged operation
- Specification
STORE REAL ADDRESS

The 64-bit real address corresponding to the second-operand virtual address is stored in the doubleword at the location designated by the first-operand address.

The virtual address specified by the B₂ and D₂ fields is translated by means of the dynamic-address-translation facility, regardless of whether DAT is on or off.

DAT is performed by using an address-space-control element that depends on the current value of the address-space-control bits, bits 16 and 17 of the PSW, as shown in the following table:

<table>
<thead>
<tr>
<th>PSW Bits 16 and 17</th>
<th>Address-Space-Control Element Used by DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Contents of control register 1</td>
</tr>
<tr>
<td>10</td>
<td>Contents of control register 7</td>
</tr>
<tr>
<td>01</td>
<td>The address-space-control element obtained by applying the access-register-translation (ART) process to the access register designated by the B₂ field</td>
</tr>
<tr>
<td>11</td>
<td>Contents of control register 13</td>
</tr>
</tbody>
</table>

The resultant 64-bit real address is stored at the first-operand location.

The translated address is not inspected for boundary alignment or for addressing or protection exceptions.

The address computations for the operands are performed according to the current addressing mode, specified by bits 31 and 32 of the current PSW.

The addresses of the region-table entry or entries, if used, and of the segment-table entry and page-table entry are treated as 64-bit addresses regardless of the current addressing mode. It is unpredictable whether the addresses of these entries are treated as real or absolute addresses.

Special Conditions

The first operand must be designated on a doubleword boundary; otherwise, a specification exception is recognized.

The operation is suppressed on all addressing exceptions.

Condition Code: The code remains unchanged.

Program Exceptions:

- Access (fetch, operand 2, except for an addressing or protection exception for the designated location; store, operand 1)
- Privileged operation
- Specification

Programming Note: STORE REAL ADDRESS is contrasted to LOAD REAL ADDRESS as follows:

- In the 24-bit or 31-bit addressing mode, LOAD REAL ADDRESS (LRA) loads bits 33-63 of the real address if bits 0-32 of the address are all zeros or recognizes a special-operation exception if bits 0-32 are not all zeros. LRA in the 64-bit addressing mode, and LOAD REAL ADDRESS (LRAG) in any addressing mode, loads bits 0-63 of the real address. STORE REAL ADDRESS stores bits 0-63 of the real address in any addressing mode.
- LOAD REAL ADDRESS, for most access-exception conditions, does not recognize the conditions as exceptions but instead sets the condition code to indicate the occurrence of the conditions. STORE REAL ADDRESS recognizes all access-exception conditions as exceptions, resulting in a program interruption.
Depending on a function code in general register 0, either an identification of the level of the configuration executing the program is placed in general register 0 or information about a component or components of a configuration is stored in a system-information block (SYSIB). When information about a component or components is requested, the information is specified by further contents of general register 0 and by contents of general register 1. The SYSIB, if any, is designated by the second-operand address.

The machine is considered to provide one, two, or three levels of configuration. The levels are:

1. The basic machine, which is the machine as if it were operating in the basic mode.

2. A logical partition, which is provided if the machine is operating in the LPAR, or logically-partitioned, mode. A logical partition is provided by the LPAR hypervisor, which is a part of the machine. A basic machine exists even when the machine is operating in the LPAR mode.

3. A virtual machine, which is provided by a virtual-machine (VM) control program that is executed either by the basic machine or in a logical partition. A virtual machine may itself execute a VM control program that provides a higher-level (more removed from the basic machine) virtual machine, which also is considered a level-3 configuration.

The terms basic mode, LPAR mode, logical partition, hypervisor, and virtual machine, and any other terms related specifically to those terms, are not defined in this publication; they are defined in the machine manuals.

A program being executed by a level-1 configuration (the basic machine) can request information about that configuration. A program being executed by a level-2 configuration (in a logical partition) can request information about the logical partition and about the underlying basic machine. A program being executed by a level-3 configuration (a virtual machine) can request information about the virtual machine and about the one or two underlying levels; a basic machine is always underlying, and a logical partition may or may not be between the basic machine and the virtual machine. When information about a virtual machine is requested, information is provided about the configuration executing the program and about any underlying level or levels of virtual machine. In any of these cases, information is provided about a level only if the level implements the instruction.

The function code determining the operation is an unsigned binary integer in bit positions 32-35 of general register 0 and is as follows:

<table>
<thead>
<tr>
<th>Function Code</th>
<th>Information Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Current-configuration-level number</td>
</tr>
<tr>
<td>1</td>
<td>Information about level 1 (the basic machine)</td>
</tr>
<tr>
<td>2</td>
<td>Information about level 2 (a logical partition)</td>
</tr>
<tr>
<td>3</td>
<td>Information about level 3 (a virtual machine)</td>
</tr>
<tr>
<td>4-15</td>
<td>None; codes are reserved</td>
</tr>
</tbody>
</table>

Invalid Function Code

The level of the configuration executing the program is called the current level. The configuration level specified by a nonzero function code is called the specified level. When the specified level is numbered higher than the current level, then the function code is called invalid, the condition code is set to 3, and no other action (including checking) is performed.

Valid Function Code

When the function code is equal to or less than the number of the current level, it is called valid. In this case, bits 36-55 of general register 0 and bits 32-47 of general register 1 must be zero; otherwise, a specification exception is recognized. Bits 0-31 of general registers 0 and 1 always are ignored.

When the function code is 0, an unsigned binary integer identifying the current configuration level (1 for basic machine, 2 for logical partition, or 3 for virtual machine) is placed in bit positions 32-35 of
general register 0, the condition code is set to 0, and no further action is performed.

When the function code is valid and nonzero, general registers 0 and 1 contain additional specifications about the information requested, as follows:

- Bit positions 56-63 of general register 0 contain an unsigned binary integer, called selector 1, that specifies a component or components of the specified configuration.
- Bit positions 48-63 of general register 1 contain an unsigned binary integer, called selector 2, that specifies the type of information requested.

The contents of general registers 0 and 1 are as follows:

```
GR 0

┌─┬────┬───────────────────┬─────────┐
│///│ FC │00000000000000000000000000000000 │Selector1│
└─┴────┴───────────────────┴─────────┘
0 32 36 56 63
GR 1

┌─┬────────────────┐
│///│0000000000000000 │Selector 2 │
└─┴────────────────┘
0 32 48 63
```

When the function code is valid and nonzero, information may be stored in a system-information block (SYSIB) beginning at the second-operand location. The SYSIB is 4K bytes and must begin at a 4K-byte boundary; otherwise, a specification exception may be recognized, depending on selector 1 and selector 2 and on whether access exceptions are recognized due to references to the SYSIB (see “Special Conditions”).

Selector 1 can have values as follows:

<table>
<thead>
<tr>
<th>Selector 1</th>
<th>Information Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None; selector is reserved</td>
</tr>
<tr>
<td>1</td>
<td>Information about the specified config-</td>
</tr>
<tr>
<td></td>
<td>uration level</td>
</tr>
<tr>
<td>2</td>
<td>Information about one or more CPUs in</td>
</tr>
<tr>
<td></td>
<td>the specified configuration level</td>
</tr>
<tr>
<td>3-255</td>
<td>None; selectors are reserved</td>
</tr>
</tbody>
</table>

When selector 1 is 1, selector 2 can have values as follows:

<table>
<thead>
<tr>
<th>Selector 2</th>
<th>Information Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None; selector is reserved</td>
</tr>
<tr>
<td>1</td>
<td>Information about the specified config-</td>
</tr>
<tr>
<td></td>
<td>uration level</td>
</tr>
<tr>
<td>2-65,535</td>
<td>None; selectors are reserved</td>
</tr>
</tbody>
</table>

When selector 1 is 2, selector 2 can have values as follows:

<table>
<thead>
<tr>
<th>Selector 2</th>
<th>Information Requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None; selector is reserved</td>
</tr>
<tr>
<td>1</td>
<td>Information about the CPU executing</td>
</tr>
<tr>
<td></td>
<td>the program in the specified configura-</td>
</tr>
<tr>
<td></td>
<td>tion level</td>
</tr>
<tr>
<td>2</td>
<td>Information about all CPUs in the spec-</td>
</tr>
<tr>
<td></td>
<td>ified configuration level</td>
</tr>
<tr>
<td>3-65,535</td>
<td>None; selectors are reserved</td>
</tr>
</tbody>
</table>

Only certain combinations of the function code, selector 1, and selector 2 are valid, as shown in Figure 10-34.

```
<table>
<thead>
<tr>
<th>Function Code</th>
<th>Selector 1</th>
<th>Selector 2</th>
<th>Information Requested about</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Current-configuration level number</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Basic-machine configuration</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>Basic-machine CPU</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>Basic-machine CPUs</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Logical-partition CPU</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>Logical-partition CPUs</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Virtual-machine CPUs</td>
</tr>
</tbody>
</table>
```

**Explanation:**
- Ignored.

**Figure 10-34. Valid Function-Code, Selector-1, and Selector-2 Combinations for STORE SYSTEM INFORMATION**

When the specified function-code, selector-1, and selector-2 combination is invalid (is other than as shown in Figure 10-34), or if it is valid but the
requested information is not available because the specified level does not implement or does not fully implement the instruction or because a necessary part of the level is uninstalled or not initialized, and provided that an exception is not recognized (see “Special Conditions”), the condition code is set to 3. When the function code is nonzero, the combination is valid, the requested information is available, and there is no exception, the requested information is stored in a system-information block (SYSIB) at the second-operand address.

Some or all of the SYSIB may be fetched before it is stored.

A SYSIB may be identified in references by means of “SYSIB fc.s1.s2,” where “fc,” “s1,” and “s2” are the values of a function code, selector 1, and selector 2, respectively.

Following sections describe the defined SYSIBs by means of figures and related text. In the figures, the offsets shown on the left are word values. “The configuration” refers to the configuration level specified by the function code (the configuration level about which information is requested).

SYSIB 1.1.1 (Basic-Machine Configuration)

SYSIB 1.1.1 has the following format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Manufacturer</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Type</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Model-Capacity Identifier</td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Sequence Code</td>
</tr>
<tr>
<td>23</td>
<td>Plant of Manufacture</td>
</tr>
<tr>
<td>25</td>
<td>Model</td>
</tr>
<tr>
<td>29</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Reserved: The contents of words 0-7, 13-15, and 29-63 are reserved and are stored as zeros. The contents of words 64-1023 are reserved and may be stored as zeros or may remain unchanged.

Manufacturer: Words 8-11 contain the 16-character (0-9 or uppercase A-Z) EBCDIC name of the manufacturer of the configuration. The name is left justified with trailing blanks if necessary.

Type: Word 12 contains the four-character (0-9) EBCDIC type number of the configuration. (This is called the machine-type number in the definition of STORE CPU ID.)

Model-Capacity Identifier: Words 16-19 contain the 16-character (0-9 or uppercase A-Z) EBCDIC model-capacity identifier of the configuration. The model-capacity identifier is left justified with trailing blanks if necessary.
**Sequence Code:** Words 20-23 contain the 16-character (0-9 or uppercase A-Z) EBCDIC sequence code of the configuration. The sequence code is right justified with leading EBCDIC zeros if necessary.

**Plant of Manufacture:** Word 24 contains the four-character (0-9 or uppercase A-Z) EBCDIC code that identifies the plant of manufacture for the configuration. The code is left justified with trailing blanks if necessary.

**Model:** When word 25 is not binary zeros, words 25-28 contain the 16-character (0-9 or uppercase A-Z) EBCDIC model identification of the configuration. The model identification is left justified with trailing blanks if necessary. (This is called the model number in programming note 4 on page 10-109 of STORE CPU ID.) When word 25 is binary zeros, the contents of words 16-19 represent both the model-capacity identifier and the model.

**Programming Notes:**

1. The fields of the SYSIB 1.1.1 are similar to those of the node descriptor described in the publication *Common I/O-Device Commands and Self Description*, SA22-7204. However, the contents of the SYSIB fields may not be identical to the contents of the corresponding node-descriptor fields because the SYSIB fields:
   - Allow more characters.
   - Are more flexible regarding the type of characters allowed.
   - Provide information that is justified differently within the field.
   - May not use the same method to determine the contents of fields such as the sequence-code field.

2. The model field in a node descriptor corresponds to the content of the STSI model field and not the STSI model-capacity-identifier field.

3. The model field specifies the model of the machine (i.e., the physical model); the model-capacity identifier field specifies a token that may be used to locate a statement of capacity or performance in the System Library publication for the model.

**SYSIB 1.2.1 (Basic-Machine CPU)**

SYSIB 1.2.1 has the following format:

<table>
<thead>
<tr>
<th>WORD</th>
<th>NAME</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-19</td>
<td>Reserved</td>
<td>Store as zeros</td>
</tr>
<tr>
<td>20-23</td>
<td>Sequence Code</td>
<td>The sequence code of the configuration.</td>
</tr>
<tr>
<td>24</td>
<td>Plant of Manufacture</td>
<td>The plant of manufacture for the configuration.</td>
</tr>
<tr>
<td>25</td>
<td>Reserved</td>
<td>Store as zeros</td>
</tr>
<tr>
<td>26</td>
<td>CPU Address</td>
<td>The CPU address of the machine.</td>
</tr>
</tbody>
</table>

**Reserved:** The contents of words 0-19, bytes 0 and 1 of word 25, and words 26-63 are reserved and stored as zeros. The contents of words 64-1023 are reserved and may be stored as zeros or may remain unchanged.

**Sequence Code:** Words 20-23 contain the 16-character (0-9 or uppercase A-Z) EBCDIC sequence code of the configuration. The code is right justified with leading EBCDIC zeros if necessary.

**Plant of Manufacture:** Word 24 contains the four-character (0-9 or uppercase A-Z) EBCDIC code that identifies the plant of manufacture for the configuration. The code is left justified with trailing blanks if necessary.

**CPU Address:** Bytes 2 and 3 of word 25 contain the CPU address by which this CPU is identified in a multiprocessing configuration. The CPU address is a 16-bit unsigned binary integer.

The CPU address is the same as is stored by STORE CPU ADDRESS when the program is executed by a machine operating in the basic mode.

**Programming Note:** Multiple CPUs in the same configuration have the same sequence code, and it is necessary to use other information, such as the CPU address, to establish a unique CPU identity. The sequence code returned for a basic-
machine CPU and a logical-partition CPU are identical and have the same value as the sequence code returned for the basic-machine configuration.

**SYSIB 1.2.2 (Basic-Machine CPUs)**

The format field in byte 0 of word 0 determines the format of the SYSIB.

When the format field has a value of zero, SYSIB 1.2.2 has a format-0 layout as follows:

```
Format-0 SYSIB 1.2.2

0 | Format | Reserved |
0-1 |        |          |
6 |        |          |
12 | Secondary CPU Capability |
13 | CPU Capability |
14 | Total CPU Count | Conf. CPU Count |
15 | SB CPU Count | Resv. CPU Count |
16 | Multiprocessing |
17 | CPU-Capability |
18 | Adjustment Factors |
```

When the format field has a value of one, SYSIB 1.2.2 has a format-1 layout as follows:

```
Format-1 SYSIB 1.2.2

0 | Format | Reserved | ACC Offset |
0-1 |        |          |           |
6 |        |          |           |
12 | Secondary CPU Capability |
13 | CPU Capability |
14 | Total CPU Count | Conf. CPU Count |
15 | SB CPU Count | Resv. CPU Count |
16 | Multiprocessing |
17 | CPU-Capability |
18 | Adjustment Factors |
N-1 | Alternate CPU Capability |
N | Alternate |
| Multiprocessing |
| CPU-Capability |
| Adjustment Factors |
N+1 | Alternate |
| Multiprocessing |
| CPU-Capability |
| Adjustment Factors |
1023 |        |
```

**Reserved:** When the format field contains a value of zero, the contents of bytes 1-3 of word 0 and words 1-6 are reserved and stored as zeros. When the format field contains a value of one, the contents of byte 1 of word 0 and words 1-6 are reserved and stored as zeros.

When fewer than 64 words are needed to contain the information for all the CPUs, the portion of the SYSIB following the adjustment-factor list in a format-0 SYSIB or the alternate-adjustment-factor list in a format-1 SYSIB, up to word 63 are reserved and are stored as zeros. The contents of words 64-1023 are reserved and may be stored as zeros or may remain unchanged.

When 64 or more words are needed to contain the information for all the CPUs, the portion of the SYSIB following the adjustment-factor list in a format-0 SYSIB or the alternate-adjustment-factor list in a format-1 SYSIB, up to word 1023 are reserved and may be stored as zeros or may remain unchanged.
Format: Byte 0 of word 0 contains an 8-bit unsigned binary integer that specifies the format of SYSIB 1.2.2.

Alternate-CPU-Capability Offset: When the format field has a value of one, bytes 2-3 of word 0 contain a 16-bit unsigned binary integer that specifies the offset in bytes of the alternate-CPU-capability field in the SYSIB.

Secondary CPU Capability: Word 7 contains a 32-bit unsigned integer that, when not zero, specifies a secondary capability that may be applied to certain types of CPUs in the configuration. There is no formal description of the algorithm used to generate this integer, except that it is the same as the algorithm used to generate the CPU capability. The integer is used as an indication of the capability of a CPU relative to the capability of other CPU models, and also relative to the capability of other CPU types within a model.

The capability value applies to each of the CPUs of one or more applicable CPU types in the configuration. That is, all CPUs in the configuration of an applicable type or types have the same capability. When the value is zero, all CPUs of any CPU type in the configuration have the same capability, as specified by the CPU capability.

The secondary CPU capability may or may not be the same value as the CPU-capability value.

The multiprocessing-CPU-capability-adjustment factors are also applicable to CPUs whose capability is specified by the secondary CPU capability.

CPU Capability: If bits 0-8 of word 8 are zero, the word contains a 32-bit unsigned binary integer (I) in the range $0 \leq I < 2^{23}$ that specifies the capability of one of the CPUs in the configuration. If bits 0-8 of word 8 are nonzero, the word contains a 32-bit binary floating point short-format number instead of a 32-bit unsigned binary integer.

Regardless of encoding, a lower value indicates a proportionally higher CPU capacity. Beyond that, there is no formal description of the algorithm used to generate this value. The value is used as an indication of the capability of the CPU relative to the capability of other CPU models.

The capability value applies to each of the non-secondary CPUs in the configuration. That is, all non-secondary CPUs in the configuration have the same capability.

Total CPU Count: Bytes 0 and 1 of word 9 contain a 16-bit unsigned binary integer that specifies the total number of CPUs in the configuration. This number includes all CPUs in the configured state, the standby state, or the reserved state.

Configured CPU Count: Bytes 2 and 3 of word 9 contain a 16-bit unsigned binary integer that specifies the number of CPUs that are in the configured state. A CPU is in the configured state when it is in the configuration and available to be used to execute programs.

Standby CPU Count: Bytes 0 and 1 of word 10 contain a 16-bit unsigned binary integer that specifies the number of CPUs that are in the standby state. A CPU is in the standby state when it is in the configuration, is not available to be used to execute programs, and can be made available by issuing instructions to place it in the configured state.

Reserved CPU Count: Bytes 2 and 3 of word 10 contain a 16-bit unsigned binary integer that specifies the number of CPUs that are in the reserved state. A CPU is in the reserved state when it is in the configuration, is not available to be used to execute programs, and cannot be made available by issuing instructions to place it in the configured state. (It may be possible to place a reserved CPU in the standby or configured state by means of manual actions.)

Multiprocessing CPU-Capability Adjustment Factors: Beginning with bytes 0 and 1 of word 11, the SYSIB contains a series of contiguous two-byte fields, each containing a 16-bit unsigned binary integer used to form an adjustment factor (fraction) for the value contained in the CPU-capability field. Such a fraction is developed by using the value (V) of the first two-byte field according to one of the following methods:

- If $V$ is in the range $0 < V \leq 100$, a denominator of 100 is indicated which produces a fraction of $V/100$.
- If $V$ is in the range $101 \leq V < 255$, a denominator of 255 is indicated which produces a fraction of $V/255$. 
• If \( V \) is in the range \( 255 \leq V < 65,536 \), a denominator of 65,536 is indicated which produces a fraction of \( V/65,536 \).

Thus, the fraction represented by each two-byte field is then developed by dividing the contents of a two-byte field by the indicated denominator.

The number of adjustment-factor fields is one less than the number of CPUs specified in the total-CPU-count field. The adjustment-factor fields correspond to configurations with increasing numbers of CPUs in the configured state. The first adjustment-factor field corresponds to a configuration with two CPUs in the configured state. Each successive adjustment-factor field corresponds to a configuration with a number of CPUs in the configured state that is one more than that for the preceding field.

**Alternate CPU Capability:** When the format field has a value of one, if bits 0-8 of word \( N \) are zero, the word contains a 32-bit unsigned binary integer \( I \) in the range \( 0 \leq I < 2^{23} \) that specifies the announced capability of one of the CPUs in the configuration. If bits 0-8 of word \( N \) are nonzero, the word contains a 32-bit binary floating point short-format number instead of a 32-bit unsigned binary integer.

Regardless of encoding, a lower value indicates a proportionally higher CPU capacity. Beyond that, there is no formal description of the algorithm used to generate this value. The value is used as an indication of the announced capability of the CPU relative to the announced capability of other CPU models.

The alternate-capability value applies to each of the CPUs in the configuration. That is, all CPUs in the configuration have the same alternate capability.

**Alternate Multiprocessing CPU-Capability Adjustment Factors:** Beginning with bytes 0 and 1 of word \( N+1 \), the SYSIB contains a series of contiguous two-byte fields, each containing a 16-bit unsigned binary integer used to form an adjustment factor (fraction) for the value contained in the alternate-CPU-capability field. Such a fraction is developed by using the value \( V \) of the first two-byte field according to one of the following methods:

- If \( V \) is in the range \( 0 < V \leq 100 \), a denominator of 100 is indicated which produces a fraction of \( V/100 \).
- If \( V \) is in the range \( 101 \leq V < 255 \), a denominator of 255 is indicated which produces a fraction of \( V/255 \).
- If \( V \) is in the range \( 255 \leq V < 65,536 \), a denominator of 65,536 is indicated which produces a fraction of \( V/65,536 \).

Thus, the fraction represented by each two-byte field is then developed by dividing the contents of a two-byte field by the indicated denominator.

The number of alternate-adjustment-factor fields is one less than the number of CPUs specified in the total-CPU-count field. The alternate-adjustment-factor fields correspond to configurations with increasing numbers of CPUs in the configured state. The first alternate-adjustment-factor field corresponds to a configuration with two CPUs in the configured state. Each successive alternate-adjustment-factor field corresponds to a configuration with a number of CPUs in the configured state that is one more than that for the preceding field.

**SYSIB 2.2.1 (Logical-Partition CPU)**

SYSIB 2.2.1 has the following format:

```
+----------------+-------------------+
| |                  |
| Reserved          |
+----------------+-------------------+
| |                  |
| Sequence Code    |
+----------------+-------------------+
| |                  |
| Plant of Manufacture |
+----------------+-------------------+
| |                  |
| LCPU ID | LCPU Address | Reserved |
+----------------+-------------------+
```

**Reserved:** The contents of words 0-19 and 26-63 are reserved and are stored as zeros. The contents of words 64-1023 are reserved and may be stored as zeros or may remain unchanged.
**Sequence Code:** Words 20-23 contain the 16-character (0-9 or uppercase A-Z) EBCDIC sequence code of the configuration. The code is right justified with leading EBCDIC zeros if necessary.

**Plant of Manufacture:** Word 24 contains the four-character (0-9 or uppercase A-Z) EBCDIC code that identifies the plant of manufacture for the configuration. The code is left justified with trailing blanks if necessary.

**Logical-CPU ID:** Bytes 0 and 1 of word 25 contain a 16-bit unsigned binary integer that can be used in conjunction with the logical-CPU address to distinguish the logical CPU from the other logical CPUs provided by the same LPAR hypervisor.

**Logical-CPU Address:** Bytes 2 and 3 of word 25 contain the logical-CPU address by which this logical CPU is identified within the level-2 configuration. The logical-CPU address is a 16-bit unsigned binary integer.

The logical-CPU-address field contains the same information as is stored by STORE CPU ADDRESS when the machine is operating in the LPAR mode.

**Programming Note:** Multiple logical CPUs in the same level-2 configuration have the same logical-CPU sequence code, and it is necessary to use other information, such as the logical-CPU address, to establish a unique logical-CPU identity. The sequence code returned for a basic-machine CPU and a logical-partition CPU are identical and have the same value as the sequence code returned for the basic-machine configuration.

**SYSIB 2.2.2 (Logical-Partition CPUs)**

SYSIB 2.2.2 has the following format:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPAR Number</td>
<td>Bytes 0 and 1 of word 8 contain a 16-bit unsigned binary integer which is the number of the level-2 configuration. This number distinguishes the configuration from all other level-2 configurations provided by the same LPAR hypervisor.</td>
</tr>
<tr>
<td>Total LCPU Count</td>
<td>The number of logical CPUs that are provided for the level-2 configuration.</td>
</tr>
<tr>
<td>SB LCPU Count</td>
<td>The number of logical CPUs that are provided using dedicated level-1 CPUs.</td>
</tr>
<tr>
<td>Logical-Partition Name</td>
<td>The unique logical-CPU identity.</td>
</tr>
<tr>
<td>Logical-Partition CAF</td>
<td>The architecture features of the logical CPU.</td>
</tr>
<tr>
<td>Ded. LCPU Count</td>
<td>The number of logical CPUs that are dedicated to this level-2 configuration.</td>
</tr>
<tr>
<td>Shr. LCPU Count</td>
<td>The number of logical CPUs that are shared between different level-2 configurations.</td>
</tr>
</tbody>
</table>

**Reserved:** The contents of words 0-7, byte 2 of word 8, words 14-17, and words 19-63 are reserved and are stored as zeros. The contents of words 64-1023 are reserved and may be stored as zeros or may remain unchanged.

**Logical-Partition Number:** Bytes 0 and 1 of word 8 contain a 16-bit unsigned binary integer which is the number of the level-2 configuration. This number distinguishes the configuration from all other level-2 configurations provided by the same LPAR hypervisor.

**Logical-CPU Characteristics (LCPUC):** The contents of byte 3 of word 8 describe the characteristics of the logical CPUs that are provided for the level-2 configuration. The bits and their meanings are as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Dedicated: When one, bit 0 indicates that one or more of the logical CPUs for this level-2 configuration are provided using level-1 CPUs that are dedicated to this level-2 configuration and are not used to provide logical CPUs for any other level-2 configuration. The number of logical CPUs that are provided using dedicated level-1 CPUs is given in the Total LCPU Count field.</td>
</tr>
</tbody>
</table>
CPUs is specified by the dedicated-LCPU-count value in bytes 0 and 1 of word 18.

When zero, bit 0 indicates that none of the logical CPUs for this level-2 configuration are provided using level-1 CPUs that are dedicated to this level-2 configuration.

1 Shared: When one, bit 1 indicates that one or more of the logical CPUs for this level-2 configuration are provided using level-1 CPUs that can be used to provide logical CPUs for other level-2 configurations. The number of logical CPUs that are provided using shared level-1 CPUs is specified by the shared-LCPU-count value in bytes 2 and 3 of word 18.

When zero, bit 1 indicates that none of the logical CPUs for this level-2 configuration are provided using shared level-1 CPUs.

2 Utilization Limit: When one, bit 2 indicates that the amount of use of the level-1 CPUs that are used to provide the logical CPUs for this level-2 configuration is limited.

When zero, bit 2 indicates that the amount of use of the level-1 CPUs that are used to provide the logical CPUs for this level-2 configuration is unlimited.

3-7 Reserved.

**Total Logical-CPU Count:** Bytes 0 and 1 of word 9 contain a 16-bit unsigned binary integer that specifies the total number of logical CPUs that are provided for this level-2 configuration. This number includes all of the logical CPUs that are in the configured state, the standby state, or the reserved state.

**Configured Logical-CPU Count:** Bytes 2 and 3 of word 9 contain a 16-bit unsigned binary integer that specifies the number of logical CPUs for this level-2 configuration that are in the configured state.

A logical CPU is in the configured state when it is in the level-2 configuration and is available to be used to execute programs.

**Standby Logical-CPU Count:** Bytes 0 and 1 of word 10 contain a 16-bit unsigned binary integer that specifies the number of logical CPUs for this level-2 configuration that are in the standby state.

A logical CPU is in the standby state when it is in the level-2 configuration, is not available to be used to execute programs, and can be made available by issuing instructions to place it in the configured state.

**Reserved Logical-CPU Count:** Bytes 2 and 3 of word 10 contain a 16-bit unsigned binary integer that specifies the number of CPUs for this level-2 configuration that are in the reserved state.

A logical CPU is in the reserved state when it is in the level-2 configuration, is not available to be used to execute programs, and cannot be made available by issuing instructions to place it in the configured state. (It may be possible to place the reserved CPU in the standby or configured state through manual actions.)

**Logical-Partition Name:** Words 11-12 contain the 8-character EBCDIC name of this level-2 configuration. The name is left justified with trailing blanks if necessary.

**Logical-Partition Capability Adjustment Factor (CAF):** Word 13 contains a 32-bit unsigned binary integer, called an adjustment factor, with a value of 1000 or less. The adjustment factor specifies the amount of the underlying level-1-configuration capability that is allowed to be used for this level-2 configuration by the LPAR hypervisor. The fraction of level-1-configuration capability is determined by dividing the CAF value by 1000.

**Dedicated Logical-CPU Count:** Bytes 0 and 1 of word 18 contain a 16-bit unsigned binary integer that specifies the number of configured-state logical CPUs for this level-2 configuration that are provided using dedicated level-1 CPUs. (See the description of bit 0 of the logical-CPU-characteristics field.)

**Shared Logical-CPU Count:** Bytes 2 and 3 of word 18 contain a 16-bit unsigned binary integer that specifies the number of configured-state logical CPUs for this level-2 configuration that are provided using shared level-1 CPUs. (See the description of bit 1 of the logical-CPU-characteristics field.)
SYSIB 3.2.2 (Virtual-Machine CPUs)

SYSIB 3.2.2 has the following format:

Reserved: The contents of words 0-6, bits 0-27 of word 7, and words 136-1023 are reserved and are stored as zeros.

Description-Block Count (DBCT): Bits 28-31 of word 7 contain a four-bit unsigned binary integer that specifies the number (up to eight) of virtual-machine description blocks that are stored in the SYSIB beginning at word 8.

Virtual-Machine Description Blocks: Words 8-135 contain from one to eight 64-byte virtual-machine description blocks, depending on the number of nested level-3 configurations, if any, and their processing characteristics.

When a level-3 configuration is provided by a virtual-machine control program and the control program is being executed by a level-3 configuration provided by another virtual-machine control program, the level-3 configurations are said to be “nested.” Level-3 configurations can be nested in this way for several levels.

The collection of nested level-3 configurations that is in the path between a program being executed by a level-3 configuration and the basic machine is called a “level-3-configuration stack.” The level-3 configurations in a stack are consecutively numbered. The level-3 configuration provided by a virtual-machine control program being executed by either a level-2 configuration or a level-1 configuration is the lowest-numbered (0) level-3 configuration in the stack. The level-3 configuration that is executing the program containing this instruction is the highest numbered (N) level-3 configuration in the stack.

If more than one virtual-machine description block is stored in words 8-135 of the SYSIB, the blocks are stored according to the following rules:

- The collection of level-3 configurations described is a contiguous subset of the total collection of level-3 configurations in the level-3-configuration stack. The subset always includes the highest-numbered level-3 configuration in the stack. One or more level-3 configurations at the bottom of the stack may not be described because STORE SYSTEM INFORMATION is not implemented by the highest of those configurations or the limit of eight description blocks would be exceeded.

- The highest-numbered level-3 configuration in the level-3-configuration stack is always described by the first description block in the SYSIB. The lowest-numbered level-3 configuration in the stack, of those that are included in the subset that is described, is described by the last description block in the SYSIB.

The contents of the SYSIB subsequent to the virtual-machine description blocks and prior to word 136 are reserved and are stored as zeros.

The virtual-machine description block has the following format:
**Reserved**: The contents of words 0 and 10-15 are reserved and are stored as zeros.

**Total Logical-CPU Count**: Bytes 0 and 1 of word 1 contain a 16-bit unsigned binary integer that specifies the total number of logical CPUs that are provided for this level-3 configuration. This number includes all of the logical CPUs that are in the configured state, the standby state, and the reserved state.

**Configured Logical-CPU Count**: Bytes 2 and 3 of word 1 contain a 16-bit unsigned binary integer that specifies the number of logical CPUs for this level-3 configuration that are in the configured state. A logical CPU is in the configured state when it is in the level-3 configuration and is available to be used to execute programs.

**Standby Logical-CPU Count**: Bytes 0 and 1 of word 2 contain a 16-bit unsigned binary integer that specifies the number of logical CPUs for this level-3 configuration that are in the standby state. A logical CPU is in the standby state when it is in the level-3 configuration, is not available to be used to execute programs, and can be made available by issuing instructions to place it in the configured state. (It may be possible to place the logical CPU in the standby or configured state through manual actions.)

**Reserved Logical-CPU Count**: Bytes 2 and 3 of word 2 contain a 16-bit unsigned binary integer that specifies the number of CPUs for this level-3 configuration that are in the reserved state. A logical CPU is in the reserved state when it is in the level-3 configuration, is not available to be used to execute programs, and cannot be made available by issuing instructions to place it in the configured state.

**Virtual-Machine Name**: Words 3-4 contain the eight-character EBCDIC name of this level-3 configuration. The name is left justified with trailing blanks if necessary.

**Virtual-Machine Capability Adjustment Factor (CAF)**: Word 5 contains a 32-bit unsigned binary integer, called an adjustment factor, with a value of 1000 or less. The adjustment factor specifies the amount of the underlying level-1-, level-2-, or level-3-configuration capability that is allowed to be used for this level-3 configuration by the virtual-machine control program. The fraction of the underlying capability is determined by dividing the CAF value by 1000.

**Control-Program Identifier**: Words 6-9 contain the 16-character EBCDIC identifier of the virtual-machine control program that provides this level-3 configuration. This identifier may include qualifiers such as version number and release level. The identifier is left justified with trailing blanks if necessary.
Special Conditions

The condition code is set to 3 if the function code in bit positions 32-35 of general register 0 is greater than the current-level number.

Bits 36-55 of general register 0 and 32-47 of general register 1 must be zero; otherwise, a specification exception is recognized.

When the function code is valid and nonzero, the following special conditions apply in an unpredictable order:

- The second operand must be designated on a 4K-byte boundary; otherwise, a specification exception is recognized.
- If the function-code, selector-1, and selector-2 combination is invalid, or if it is valid but the requested information is not available, the condition code is set to 3.

The priority of the recognition of exceptions and condition codes is shown in Figure 10-35.

Resulting Condition Code:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Requested configuration-level number placed in general register 0 or requested SYSIB information stored</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>Requested SYSIB information not available</td>
</tr>
</tbody>
</table>

Program Exceptions:

- Access (store, operand 2, only if function code nonzero)
- Privileged operation
- Specification

---

1. Exceptions with the same priority as the priority of program-interruption conditions for the general case.
3. Operation exception if the store-system-information facility is not installed.
4. Privileged-operation exception for privileged instruction.
5. Condition code 3 due to function code greater than current-level number.
6. Specification exception due to bits 36-55 of general register 0 or bits 32-47 of general register 1 not zero.
7. Condition code 0 due to function code 0.
8. Specification exception due to second-operand address not designating a 4K-byte boundary.
9. Condition code 3 due to invalid function-code, selector-1, and selector-2 combination or requested information not available.
10. Access exceptions (store) for system-information block.
11. Condition code 0 due to information stored in system-information block.

---

Figure 10-35. Priority of Execution: STORE SYSTEM INFORMATION

Programming Note: The storage-operand references for STORE SYSTEM INFORMATION may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)
STORE THEN AND SYSTEM MASK

STNSM  D₁(B₁),I₂  [SI]

<table>
<thead>
<tr>
<th>'AC'</th>
<th>I₂</th>
<th>B₁</th>
<th>D₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits 0-7 of the current PSW are stored at the first-operand location. Then the contents of bit positions 0-7 of the current PSW are replaced by the logical AND of their original contents and the second operand.

Special Conditions

The operation is suppressed on addressing and protection exceptions.

Condition Code:  The code remains unchanged.

Program Exceptions:
- Access (store, operand 1)
- Privileged operation

Programming Note:  STORE THEN AND SYSTEM MASK permits the program to set selected bits in the system mask to zeros while retaining the original contents for later restoration. For example, it may be necessary that a program, which has no record of the present status, disable program-event recording for a few instructions.

STORE THEN OR SYSTEM MASK

STOSM  D₁(B₁),I₂  [SI]

<table>
<thead>
<tr>
<th>'AD'</th>
<th>I₂</th>
<th>B₁</th>
<th>D₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bits 0-7 of the current PSW are stored at the first-operand location. Then the contents of bit positions 0-7 of the current PSW are replaced by the logical OR of their original contents and the second operand.

Special Conditions

The operation is suppressed on addressing and protection exceptions.

Condition Code:  The code remains unchanged.

Program Exceptions:
- Access (store, operand 1)
- Privileged operation
- Specification

Programming Note:  STORE THEN OR SYSTEM MASK permits the program to set selected bits in the system mask to ones while retaining the original contents for later restoration. For example, the program may enable the CPU for I/O interruptions without having available the current status of the external-mask bit.

STORE USING REAL ADDRESS

STURA  R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>'B246'</th>
<th>//////////</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

STURG  R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>'B925'</th>
<th>//////////</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

For STORE USING REAL ADDRESS (STURA), bits 32-63 of general register R₁ are stored in the word at the real-storage location addressed by the contents of general register R₂. For STORE USING REAL ADDRESS (STURG), bits 0-63 of general register R₁ are stored in the doubleword at that real-storage location.

In the 24-bit addressing mode, bits 40-63 of general register R₂ designate the real-storage location, and bits 0-39 of the register are ignored. In the 31-bit addressing mode, bits 33-63 of...
general register R2 designate the real-storage location, and bits 0-33 of the register are ignored. In the 64-bit addressing mode, bits 0-63 of general register R2 designate the real-storage location.

Because it is a real address, the address designating the storage word or doubleword is not subject to dynamic address translation.

**Special Conditions**

The contents of general register R2 must designate a location on a word boundary for STURA or on a doubleword boundary for STURG; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Addressing (address specified by general register R2)
- Privileged operation
- Protection (store, operand 2, key-controlled protection and low-address protection)
- Specification

**TEST ACCESS**

\[
\text{TAR } R_1, R_2 \quad [\text{RRE}]
\]

<table>
<thead>
<tr>
<th>'B24C'</th>
<th>/ / / / / /</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The access-list-entry token (ALET) in access register R1 is tested for exceptions recognized during access-register translation (ART). The extended authorization index (EAX) used is bits 32-47 of general register R2. The ALET is also tested for whether it designates the dispatchable-unit access list or the primary-space access list and for whether it is 00000000 or 00000001 hex.

When R1 is 0, the actual contents of access register 0 are used in ART, instead of the 00000000 hex that is usually used.

Bits 0-31 and 48-63 of general register R2 are ignored.

The operation does not depend on the translation mode — bits 5, 16, and 17 of the PSW are ignored.

When the ALET specified by means of the R1 field is other than 00000000 or 00000001 hex, the ART process is applied to the ALET. The EAX specified by means of the R2 field is called the effective EAX, and it is the EAX which is used by ART. When a condition exists that would normally cause one of the exceptions shown in the following table, the instruction is completed by setting condition code 3.

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALET specification</td>
<td>ALET bits 0-6 not all zeros</td>
</tr>
<tr>
<td>ALEN translation</td>
<td>Access-list entry (ALE) outside list or invalid (bit 0 is one)</td>
</tr>
<tr>
<td>ALE sequence</td>
<td>ALE sequence number (ALESN) in ALET not equal to ALESN in ALE</td>
</tr>
<tr>
<td>ASTE validity</td>
<td>ASN-second-table entry (ASTE) invalid (bit 0 is one)</td>
</tr>
<tr>
<td>ASTE sequence</td>
<td>ASTE sequence number (ASTESN) in ALE not equal to ASTESN in ASTE</td>
</tr>
<tr>
<td>Extended authority</td>
<td>ALE private bit not zero, ALE authorization index (ALEAX) not equal to effective EAX, and secondary bit selected by effective EAX either outside authority table or zero</td>
</tr>
</tbody>
</table>

When ART is completed without one of the above conditions being present, the instruction is completed by setting condition code 1 or 2, depending on whether the effective access list is the dispatchable-unit access list or the primary-space access list, respectively. The effective access list is the dispatchable-unit access list if bit 7 of the ALET is zero, or it is the primary-space access list if bit 7 is one. ART, including the obtaining of the effective access-list designation, is described in "Access-Register-Translation Process" on page 5-57.

When the ALET is 00000000 hex, the instruction is completed by setting condition code 0. When
the ALET is 00000001 hex, the instruction is completed by setting condition code 3.

Special Conditions

An addressing exception is recognized when the address used by ART to fetch the effective access-list designation or the ALE, ASTE, or authority-table entry designates a location which is not available in the configuration.

The operation is suppressed on all addressing exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-36 on page 10-127.

Resulting Condition Code:

0 Access-list-entry token (ALET) is 00000000 hex
1 ALET designates the dispatchable-unit access list and does not cause exceptions in access-register translation (ART)
2 ALET designates the primary-space access list and does not cause exceptions in ART
3 ALET is 00000001 hex or causes exceptions in ART

Program Exceptions:

- Addressing (effective access-list designation, access-list entry, ASN-second-table entry, or authority-table entry)

Programming Notes:

1. TEST ACCESS permits a called program to check whether an ALET passed from the calling program is authorized for use by means of the calling program's EAX. The calling program's EAX can be obtained from the last linkage-stack state entry by means of EXTRACT STACKED STATE. The called program can thus avoid performing an operation for the calling program, through the use of the called program's EAX, which the calling program is not authorized to perform by means of its own EAX.

2. When an ALET equal to 00000000 hex is passed during a program linkage performed by PROGRAM CALL with space switching (PC-ss), and the ALET conceptually designates the calling program's primary address space and the called program's secondary address space, the ALET must be changed to 00000001 hex before it is used by the called program. Condition code 0 of TEST ACCESS indicates a 00000000 hex ALET so that the ALET can be changed to 00000001 hex by the called program.

3. PROGRAM CALL to current primary (PC-cp) sets the secondary address space equal to the primary address space. PC-ss sets the secondary address space equal to the calling program's primary address space, except that stacking PC-ss sets it equal to the called program's primary address space when the secondary-ASN control in the entry-table entry used is one. In all these cases, a passed 00000001 hex ALET that conceptually designates the calling program's secondary address space is not usable by the called program, even after any transformation (unless the operation was PC-cp and the calling program's PASN and SASN are equal). This is why TEST ACCESS sets condition code 3 when the tested ALET is 00000001 hex.

4. After a PC-ss, a passed ALET that conceptually designates an entry in the primary-space access list of the calling program is not usable by the called program. This is why TEST ACCESS sets condition code 2, instead of condition code 1, when the tested ALET designates the primary-space access list.

5. The control program may manage the ASN-second-table entry in a way that causes a correctable ASTE-validity or ASTE-sequence exception situation to exist; that is, a situation which, if it were to cause a program interruption during access-register translation, would be corrected by the control program so that access-register translation could be completed successfully. In this case, the program should not use TEST ACCESS directly but should instead use a control-program service that uses TEST ACCESS and that corrects the situation, if possible, when condition code 3 is set. MVS/ESA provides the TESTART macro instruction for use instead of the direct use of TEST ACCESS.
1. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


3. Condition code 0 due to access-list-entry-token (ALET) being 00000000 hex.

4. Condition code 3 due to ALET being 00000001 hex or ALET bits 0-6 not being all zeros.

5. Addressing exception for access to effective access-list designation.

6. Condition code 3 due to access-list entry (ALE) being outside the list.

7. Accessing exception for access to ALE.

8. Condition code 3 due to ALE being invalid (bit 0 is 1) or access-list-entry sequence number (ALESN) in the ALET not being equal to the ALESN in the ALE.

9. Addressing exception for access to ASN-second-table entry (ASTE).

10. Condition code 3 due to ASTE being invalid (bit 0 is one) or ASTE sequence number (ASTESN) in the ALE not being equal to the ASTESN in the ASTE.

11. Addressing exception for access to authority-table entry.

12. Condition code 3 due to authority-table entry being outside table.

13. Addressing exception for access to authority-table entry.

14. Condition code 3 due to ALE private bit not being zero, ALE authorization index (ALEAX) not being equal to effective extended authorization index (EAX), and secondary bit selected by effective EAX being zero.

15. Condition code 1 if ALET bit 7 is zero; otherwise, condition code 2.

The block tested is addressed by the contents of general register R2. The contents of general register R1 are ignored.

A complete testing operation is necessarily performed only when the initial contents of bit positions 32-63 of general register 0 are zero in the 24-bit or 31-bit addressing mode, or the initial contents of bit positions 0-63 of that register are zero in the 64-bit addressing mode. In the 24-bit or 31-bit addressing mode, the contents of bit positions 32-63 of general register 0 are set to zero at the completion of the operation, and bits 0-31 of...
the register always are ignored and remain unchanged. In the 64-bit addressing mode, the contents of bit positions 0-63 of the register are set to zero at the completion of the operation.

If the block is found to be usable, the 4K bytes of the block are cleared to zeros, the contents of the storage key are unpredictable, and condition code 0 is set. If the block is found to be unusable, the data and the storage key are set, as far as is possible by the model, to a value such that subsequent fetches to the area do not cause a machine-check condition, and condition code 1 is set.

In the 24-bit addressing mode, bits 40-51 of general register $R_{two}$ designate a 4K-byte block in real storage, and bits 0-39 and 52-63 of the register are ignored. In the 31-bit addressing mode, bits 33-51 of the register designate the block, and bits 0-32 and 52-63 are ignored. In the 64-bit addressing mode, bits 0-51 of the register designate the block, and bits 52-63 are ignored.

The address of the block is a real address, and the accesses to the block designated by the second-operand address are not subject to key-controlled, access-list-controlled, and page protection. Low-address protection does apply. The operation is terminated on addressing and protection exceptions. If termination occurs, the condition code and the contents of bit positions 32-63 of general register 0 are unpredictable in the 24-bit or 31-bit addressing mode, or the condition code and bits 0-63 of the register are unpredictable in the 64-bit addressing mode. The contents of the storage block and its associated storage key are not changed when these exceptions occur.

Depending on the model, the test for usability may be performed (1) by alternately storing and reading out test patterns to the data and storage key in the block or (2) by reference to an internal record of the usability of the blocks which are available in the configuration, or (3) by using a combination of both mechanisms.

In models in which an internal record is used, the block is indicated as unusable if a solid failure has been previously detected, or if intermittent failures in the block have exceeded the threshold implemented by the model. In such models, depending on the criteria, attempts to store may or may not occur. Thus, if block 0 is not usable, and no store occurs, low-address protection may or may not be indicated.

In models in which test patterns are used, TEST BLOCK may be interruptible. When an interruption occurs after a unit of operation, other than the last one, the condition code is unpredictable, and the contents of bit positions 32-63 of general register 0 may contain a record of the state of intermediate steps in the 24-bit or 31-bit addressing mode, or the contents of bit positions 0-63 may contain that record in the 64-bit addressing mode. When execution is resumed after an interruption, the condition code is ignored, but the record in general register 0 may be used to determine the resumption point.

If (1) TEST BLOCK is executed with an initial value other than zero in bit positions 32-63 of general register 0 in the 24-bit or 31-bit addressing mode or bit positions 0-63 in the 64-bit addressing mode, or (2) the interrupted instruction is resumed after an interruption with a value in bit positions 32-63 or 0-63 (depending on the addressing mode) of general register 0 or a value in the storage block or its associated storage key other than the corresponding value which was present at the time of the interruption, or (3) the block or its associated storage key is accessed by another CPU or by the channel subsystem during the execution of the instruction, then the contents of the storage block, its associated storage key, and bit positions 32-63 or 0-63 of general register 0 are unpredictable, along with the resultant condition-code setting.

Invalid checking-block-code errors initially found in the block or encountered during the test do not normally result in machine-check conditions. The test-block function is implemented in such a way that the frequency of machine-check interruptions due to the instruction execution is not significant. However, if, during the execution of TEST BLOCK for an unusable block, that block is accessed by another CPU (or by the channel subsystem), error conditions may be reported both to this CPU and to the other CPU (or to the channel subsystem).

A serialization function is performed before the block is accessed and again after the operation is completed (or partially completed).
The priority of the recognition of exceptions and condition codes is shown in Figure 10-37 on page 10-129.

**Resulting Condition Code:**

0  Block usable
1  Block not usable
2  --
3  --

**Program Exceptions:**

- Addressing (fetch and store, operand 2)
- Privileged operation
- Protection (store, operand 2, low-address protection only)

1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.


7.B Privileged-operation exception.

8. Addressing exception due to block not being available in the configuration.*

9.A Condition code 1, block not usable.

9.B Protection exception due to low-address protection.*

10. Condition code 0, block usable and set to zeros.

**Explanation:**

* The operation is terminated on addressing and protection exceptions, and the condition code may be unpredictable.

---

**Program Notes:**

1. The execution of TEST BLOCK on most models is significantly slower than that of the MOVE LONG instruction with padding; therefore, the instruction should not be used for the normal case of clearing storage.

2. The program should use TEST BLOCK at initial program loading and as part of the vary-storage-online procedure to determine if blocks of storage exist which should not be used.

3. The program should use TEST BLOCK when an uncorrected error is reported in either the data or storage key of a block. This is because in the execution of TEST BLOCK the attempt is made, as far as is possible on the model, to leave the contents of a block in a state such that subsequent prefetches or unintended references to the block do not cause machine-check conditions. The program may use the resulting condition code in this case to determine if the block can be reused. (The block could be indicated as usable if, for example, the error were an externally generated error or an indirect storage error.) This procedure should be followed regardless of whether the indirect-storage-error indication is reported.

4. The model may or may not be successful in removing the errors from a block when TEST BLOCK is executed. The program therefore should take every reasonable precaution to avoid referencing an unusable block. For example, the program should not place the page-frame real address of an unusable block in an attached and valid page-table entry.

5. On some models, machine checks may be reported for a block even though the block is not referenced by the program. When a machine check is reported for a storage-key error in a block which has been marked as unusable by the program, it is possible that SET STORAGE KEY EXTENDED may be more effective than TEST BLOCK in validating the storage key.

6. The storage-operand references for TEST BLOCK may be multiple-access references. (See "Storage-Operand Consistency" on page 5-98)

---

**TEST PROTECTION**

```
TPROT  D1(B1),D2(B2)  [SSE]

<table>
<thead>
<tr>
<th>'E501'</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>
```

The location designated by the first-operand address is tested for protection exceptions by using the access key specified by bits 56-59 of the second-operand address.

The second-operand address is not used to address data; instead, bits 56-59 of the address...
form the access key to be used in testing. Bits 0-55 and 60-63 of the second-operand address are ignored.

The first-operand address is a logical address. When the CPU is in the access-register mode (when DAT is on and PSW bits 16 and 17 are 01 binary), the first-operand address is subject to translation by means of both the access-register-translation (ART) and the dynamic-address-translation (DAT) processes. ART applies to the access register designated by the $B_1$ field, and it obtains the address-space-control element to be used by DAT. When DAT is on but the CPU is not in the access-register mode, the first-operand address is subject to translation by DAT. In this case, DAT uses the address-space-control element contained in control register 1, 7, or 13 when the CPU is in the primary-space, secondary-space, or home-space mode, respectively. When DAT is off, the first-operand address is a real address not subject to translation by either ART or DAT.

When the CPU is in the access-register mode and an address-space-control element cannot be obtained by ART because of a condition that would normally cause one of the exceptions shown in the following table, the instruction is completed by setting condition code 3.

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALET specification</td>
<td>Access-list-entry-token (ALET) bits 0-6 not all zeros</td>
</tr>
<tr>
<td>ALEN translation</td>
<td>Access-list entry (ALE) outside list or invalid (bit 0 is one)</td>
</tr>
<tr>
<td>ALE sequence</td>
<td>ALE sequence number (ALESN) in ALET not equal to ALESN in ALE</td>
</tr>
<tr>
<td>ASTE validity</td>
<td>ASN-second-table entry (ASTE) invalid (bit 0 is one)</td>
</tr>
<tr>
<td>ASTE sequence</td>
<td>ASTE sequence number (ASTESN) in ALE not equal to ASTESN in ASTE</td>
</tr>
<tr>
<td>Extended authority</td>
<td>ALE private bit not zero, ALE authorization index (ALEAX) not equal to extended authorization index (EAX), and secondary bit selected by EAX either outside authority table or zero</td>
</tr>
</tbody>
</table>

When the access register contains 00000000 hex or 00000001 hex, ART obtains the address-space-control element from control register 1 or 7, respectively, without accessing the access list. When the $B_1$ field designates access register 0, ART treats the access register as containing 00000000 hex and does not examine the actual contents of the access register.

When ART is completed successfully, the operation is continued through the performance of DAT.

When DAT is on and the first-operand address cannot be translated because of a condition that would normally cause one of the exceptions shown in the following table, the instruction is completed by setting condition code 3.
When translation of the first-operand address can be completed, or when DAT is off, the storage key for the block designated by the first-operand address is tested against the access key specified in bit positions 56-59 of the second-operand address, and the condition code is set to indicate whether store and fetch accesses are permitted, taking into consideration all applicable protection mechanisms. Thus, for example, if low-address protection is active and the first-operand effective address is in the range 0-511 or 4096-4607, then a store access is not permitted. Access-list-controlled protection, page protection, storage-protection override, and fetch-protection override also are taken into account.

The contents of storage, including the change bit, are not affected. Depending on the model, the reference bit for the first-operand address may be set to one, even for the case in which the location is protected against fetching.

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCE type</td>
<td>Address-space-control element (ASCE) being used is a region-second-table designation, and bits 0-10 of first-operand address not all zeros; ASCE is a region-third-table designation, and bits 0-21 of first-operand address not all zeros; or ASCE is a segment-table designation, and bits 0-32 of first-operand address not all zeros.</td>
</tr>
<tr>
<td>Region first translation</td>
<td>Region-first-table entry outside table or invalid.</td>
</tr>
<tr>
<td>Region second translation</td>
<td>Region-second-table entry outside table or invalid.</td>
</tr>
<tr>
<td>Region third translation</td>
<td>Region-third-table entry outside table or invalid.</td>
</tr>
<tr>
<td>Segment translation</td>
<td>Segment-table entry outside table or invalid.</td>
</tr>
<tr>
<td>Page translation</td>
<td>Page-table entry invalid</td>
</tr>
</tbody>
</table>

When translation of the first-operand address can be completed, or when DAT is off, the storage key for the block designated by the first-operand address is tested against the access key specified in bit positions 56-59 of the second-operand address, and the condition code is set to indicate whether store and fetch accesses are permitted, taking into consideration all applicable protection mechanisms. Thus, for example, if low-address protection is active and the first-operand effective address is in the range 0-511 or 4096-4607, then a store access is not permitted. Access-list-controlled protection, page protection, storage-protection override, and fetch-protection override also are taken into account.

The contents of storage, including the change bit, are not affected. Depending on the model, the reference bit for the first-operand address may be set to one, even for the case in which the location is protected against fetching.

**Special Conditions**

When the CPU is in the access-register mode, an addressing exception is recognized when the address used by ART to fetch the effective access-list designation or the ALE, ASTE, or authority-table entry designates a location which is not available in the configuration.

When DAT is on, an addressing exception is recognized when the address of the region-table entry or entries, segment-table entry, or page-table entry or the operand real address after translation designates a location which is not available in the configuration. Also, a translation-specification exception is recognized when a region-table entry or the segment-table entry or page-table entry has a format error, that is, when any of the reasons listed in “Translation-Specification Exception” on page 6-36 applies. When DAT is off, only the addressing exception due to the operand real address applies.

For all of the above cases, the operation is suppressed.

**Resulting Condition Code:**

- 0 Fetching permitted; storing permitted
- 1 Fetching permitted; storing not permitted
- 2 Fetching not permitted; storing not permitted
- 3 Translation not available

**Program Exceptions:**

- Addressing (effective access-list designation, access-list entry, ASN-second-table entry, authority-table entry, region-table entry, segment-table entry, page-table entry, or operand 1)
- Privileged operation
- Translation specification

**Programming Notes:**

1. TEST PROTECTION permits a program to check the validity of an address passed from a calling program without incurring program exceptions. The instruction sets a condition code to indicate whether fetching or storing is permitted at the location designated by the first-operand address of the instruction. The instruction takes into consideration all of the protection mechanisms in the machine: access-list controlled, page, key-controlled, and low-address protection, storage-protection.
override, and fetch-protection override. Additionally, since ASCE-type, region-translation, segment-translation, and page-translation-exception conditions may be a program substitute for a protection violation, these conditions are used to set the condition code rather than cause a program exception.

When the CPU is in the access-register mode, TEST PROTECTION additionally permits the program to check the usability of an access-list-entry token (ALET) in an access register without incurring program exceptions. The ALET is checked for validity (absence of an ALET-specification, ALEN-translation, and ALE-sequence-exception condition) and for being authorized for use by the program (absence of an ASTE-validity, ASTE-sequence, and extended-authority-exception condition).

An ASCE-type-exception condition also causes setting of the condition code.

2. See the programming notes under SET PSW KEY FROM ADDRESS for more details and for an alternative approach to testing validity of addresses passed by a calling program. The approach using TEST PROTECTION has the advantage of a test which does not result in interruptions; however, the test and use are separated in time and may not be accurate if the possibility exists that the storage key of the location in question can change between the time it is tested and the time it is used.

3. In the handling of dynamic address translation, TEST PROTECTION is similar to LOAD REAL ADDRESS in that the instructions do not cause ASCE-type, region-translation, segment-translation, and page-translation exceptions. Instead, these exception conditions are indicated by means of a condition-code setting. Similarly, access-register translation sets a condition code for certain exception conditions when performed during either of the two instructions. Conditions which result in condition codes 1, 2, and 3 for LOAD REAL ADDRESS result in condition code 3 for TEST PROTECTION. The instructions also differ in several other respects. The first-operand address of TEST PROTECTION is a logical address and thus is not subject to dynamic address translation when DAT is off. The second-operand address of LOAD REAL ADDRESS is a virtual address which is always translated.

Access-register translation applies to TEST PROTECTION only when the CPU is in the access-register mode (DAT is on), whereas it applies to LOAD REAL ADDRESS when PSW bits 16 and 17 are 01 binary regardless of whether DAT is on or off. When condition code 3 is set because of an exception condition in access-register translation, LOAD REAL ADDRESS, but not TEST PROTECTION, returns in a general register the program-interruption code assigned to the exception.

### TRACE

**TRACE**

```
TRACE  R1,R3,D2(B2)  [RS]
```

<table>
<thead>
<tr>
<th>'99'</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TRACG**

```
TRACG R1,R3,D2(B2)  [RSY]
```

<table>
<thead>
<tr>
<th>'EB'</th>
<th>R1</th>
<th>R3</th>
<th>B2</th>
<th>DL2</th>
<th>DH2</th>
<th>'OF'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When explicit tracing is on (bit 63 of control register 12 is one), the second operand, which is a 32-bit word in storage, is fetched, and bit 0 of the word is examined. If bit 0 of the second operand is zero, a trace entry is formed at the real-storage location designated by control register 12.

If explicit tracing is off (bit 63 of control register 12 is zero), or if bit 0 of the second operand is one, no trace entry is formed, and no trace exceptions are recognized.

The displacement for TRACE is treated as a 12-bit unsigned binary integer. The displacement for TRACG is treated as a 20-bit signed binary integer.

The trace entry is composed of an entry-type identifier, a count of the number of general registers whose partial or entire contents are placed in the entry, a field whose contents indicate whether the entry was formed by TRACE (TRACE) or TRACE (TRACG), selected bits of the TOD clock, the second operand, and the partial or entire contents
of a range of general registers. For TRACE (TRACE), bits 16-63 of the TOD clock and bits 32-63 of the general registers are placed in the trace entry. For TRACE (TRACG), bits 0-79 of the clock and bits 0-63 of the registers are placed in the entry.

The general registers are stored in ascending order of their register numbers, starting with general register R1 and continuing up to and including general register R3, with general register 0 following general register 15. The trace table and the trace-entry formats are described in "Tracing" on page 4-10.

When a trace entry is made, a serialization and checkpoint-synchronization function is performed before the operation begins and again after the operation is completed. However, it is unpredictable whether or not a store into a trace-table entry from which a subsequent instruction is fetched will be observed by the CPU that performed the store.

**Special Conditions**

A privileged-operation exception is recognized in the problem state, even when explicit tracing is off or bit 0 of the second operand is one.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized. It is unpredictable whether the specification exception is recognized when explicit tracing is off.

It is unpredictable whether access exceptions are recognized for the second operand when explicit tracing is off.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2)
- Privileged operation
- Specification
- Trace

**Programming Note:** Bits 1-15 of the second operand are reserved for model-dependent functions and should therefore be set to zeros.

### TRAP

<table>
<thead>
<tr>
<th>TRAP2</th>
<th>[E]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'01FF'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TRAP4</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B2FF'</td>
<td>B2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

A trap operation is performed if the CPU is in the primary-space or access-register mode and the TRAP-enabled bit in byte 47 of the dispatchable-unit control table (DUCT) is one. Otherwise, a special-operation exception is recognized.

The trap operation obtains a trap-control-block address from the DUCT and then a trap-save-area address and a trap-program address from the trap control block. State information is stored in the trap save area. Then the trap-control-block address is loaded into general register 15. Finally, the current PSW is updated by setting the basic-addressing-mode bit to one (which will leave the addressing mode as either the 31-bit mode or the 64-bit mode or will change the addressing mode from the 24-bit mode to the 31-bit mode) and the address-space-control bits to zeros (primary-space mode) and by replacing the instruction address with the trap-program address. Compatibility with the ESA/390 operation of TRAP optionally is provided.

For TRAP4, the second-operand address is not used to address data; instead, bits 33-63 of the address are stored in the trap save area.

### Dispatchable-Unit Control Table

Bytes 44-47 (word 10) of the dispatchable-unit control table (DUCT) are used by this instruction. The contents of those bytes are as follows:

<table>
<thead>
<tr>
<th>DUCT Bytes 44-47</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap-Control-Block Address</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

The fields in bytes 44-47 of the DUCT are allocated as follows:
**Trap-Control-Block Address:** Bits 1-28, with three zeros appended on the right, form the 31-bit home virtual address of the trap control block. This address is treated as a 31-bit home virtual address regardless of the current addressing mode and regardless of the current value of the address-space-control bits. This address, with a zero appended on the left, is placed in bit positions 32-63 of general register 15 after the contents of that register have been saved in the trap save area. If the current addressing mode is the 64-bit mode, bits 0-31 of general register 15 are set to zeros.

**TRAP-Enabled Bit (E):** Bit 31 specifies, when one, that the trap operation is to be performed. TRAP recognizes a special-operation exception if bit 31 is zero.

Bits 0, 29, and 30 of bytes 44-47 are ignored, but they should be zeros to permit possible future extensions.

**Trap Control Block**

The trap control block is 64 bytes aligned on a doubleword boundary. The format of the trap control block is:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The fields in the trap control block are allocated as follows:

- **PSW Control (P):** Bit 13 of bytes 0-3 controls the allowed value of bit 31 of the current PSW and how bits 12 and 33-127 of the current PSW are stored in the PSW-values field in the trap save area. When bit 13 is zero:
  - Bit 31 of the current PSW, the extended-addressing-mode bit, must be zero; otherwise, a special-operation exception is recognized.
  - A one is stored in bit position 12 of the PSW-values field even though bit 12 of the current PSW is zero.
  - Bits 97-127 of the current PSW are stored in bit positions 33-63 of the PSW-values field, bits 33-96 of the current PSW are not stored, and zeros are stored in bit positions 64-127 of the PSW-values field.

When bit 13 is one:
  - Bit 31 of the current PSW may be zero or one.
  - Bit 12 of the current PSW is stored in bit position 12 of the PSW-values field.
  - Bits 64-127 of the current PSW are stored in bit positions 64-127 of the PSW-values field.

- **General-Registers Control (R):** Bit 14 of bytes 0-3 controls how the contents of the general registers are stored in the general-registers 0-15 field in the trap save area. When R is zero, bits 32-63 of the general registers are stored in consecutive four-byte locations beginning at the beginning of the general-registers 0-15 field, bits 0-31 of the registers are not stored, and the last 64 bytes of the general-registers 0-15 field remain unchanged. When R is one, bits 0-63 of the general registers are stored in consecutive eight-byte locations in the general-registers 0-15 field.

- **Trap-Save-Area Address:** Bits 1-28 of bytes 12-15, with three zeros appended on the right, form the 31-bit home virtual address of the trap save area. This address is treated as a 31-bit home virtual address regardless of the current addressing mode and regardless of the current value of the address-space-control bits. Bits 0 and 29-31 of bytes 12-15 are ignored.

- **Trap-Program Address:** Bits 1-31 of bytes 20-23 form the 31-bit primary virtual address of the trap program. This address is treated as a 31-bit primary virtual address regardless of the current addressing mode.

Bit positions 0-12 and 15-31 of bytes 0-3 and bytes 4-11, 16-19, and 32-63 of the trap control block are reserved and should contain zeros. Bytes 24-31 are available for use by programming.
**Trap Save Area**

The trap save area is 256 bytes aligned on a doubleword boundary.

The trap operation stores information into the trap save area as follows:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Trap Flags</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Reserved (Zeros Stored)</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>Bits 33-63 of Second-Op Address of TRAP4</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>Access Register 15</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>PSW Values</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
<td>PSW Values</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>PSW Values</td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
<td>PSW Values</td>
</tr>
<tr>
<td>20</td>
<td>32</td>
<td>General Registers 0-15</td>
</tr>
<tr>
<td>24</td>
<td>36</td>
<td>General Registers 0-15</td>
</tr>
<tr>
<td>98</td>
<td>152</td>
<td>General Registers 0-15</td>
</tr>
<tr>
<td>9C</td>
<td>156</td>
<td>General Registers 0-15</td>
</tr>
<tr>
<td>A0</td>
<td>160</td>
<td>General Registers 0-15</td>
</tr>
<tr>
<td>A4</td>
<td>164</td>
<td>General Registers 0-15</td>
</tr>
<tr>
<td>A8</td>
<td>168</td>
<td>Reserved (Unchanged)</td>
</tr>
<tr>
<td>AC</td>
<td>172</td>
<td>Reserved (Unchanged)</td>
</tr>
<tr>
<td>F8</td>
<td>248</td>
<td>Reserved (Unchanged)</td>
</tr>
<tr>
<td>FC</td>
<td>252</td>
<td>Reserved (Unchanged)</td>
</tr>
</tbody>
</table>

The fields in the trap save area are allocated as follows:

**Trap Flags:** Information identifying the instruction(s) causing the trap operation is stored in byte positions 0-3. The detailed format of bytes 0-3 is as follows:

<table>
<thead>
<tr>
<th>Flag Bits</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TRAP was target of EXECUTE</td>
</tr>
<tr>
<td>1</td>
<td>TRAP is TRAP4 (not TRAP2)</td>
</tr>
<tr>
<td>2-12</td>
<td>Reserved, zeros stored</td>
</tr>
<tr>
<td>13-14</td>
<td>Instruction-length code (ILC)</td>
</tr>
<tr>
<td>15-31</td>
<td>Reserved, zeros stored</td>
</tr>
</tbody>
</table>

Bit 0 of bytes 0-3 is set to one if TRAP was the target of an EXECUTE instruction.

Bit 1 of bytes 0-3 is set to one if TRAP is TRAP4 (not TRAP2).

Bits 13 and 14 are the instruction-length code (ILC) that specifies the length of the TRAP instruction, or the length of the EXECUTE instruction if TRAP was the target of EXECUTE.

Bits 2-12 and 15-31 are reserved and are stored as zeros.

**Bits 33-63 of Second-Operand Address of TRAP4:** For TRAP4, bits 33-63 of the second-operand address, generated under the control of the current addressing mode and with a zero appended on the left, are stored in byte positions 8-11. Only bits 33-63 of the second-operand address are stored even when the current addressing mode is the 64-bit mode. For TRAP2, all zeros are stored in byte positions 8-11.

**Access Register 15:** The contents of access register 15 are stored in byte positions 12-15.

**PSW Values:** The following description applies when the PSW control, bit 13 of bytes 0-3 of the trap control block, is one.

Certain information from the current PSW is stored in byte positions 16-31. The PSW has the following format:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>1</td>
<td>Unpredictable</td>
</tr>
</tbody>
</table>
In summary, bits 0, 2-4, 12, 24-30, and 33-63 are zero, bits 1, 5-11, and 13 are unpredictable, and the other bits are set with variable information from the PSW.

The wait-state, problem-state, address-space-control, condition-code, program-mask, extended-addressing-mode, and basic-addressing-mode values specify the state of the CPU before the TRAP instruction was executed. The instruction-address value is the updated instruction address, which is the address of the instruction following TRAP, or the address of the instruction following EXECUTE if TRAP was the target of EXECUTE.

When the PSW control in the trap control block is zero, the operation is as described above except as follows:

- Bit 31 of the current PSW must be zero; otherwise, a special-operation exception is recognized.
- A one is stored in bit position 12 of bytes 16-31.
- Bits 97-127 of the current PSW are stored in bit positions 33-63 of bytes 16-31, bits 33-96 of the current PSW are not stored, and zeros are stored in bit positions 64-127 of bytes 16-31 (bytes 24-31).

In this case, bytes 16-23 have the format of an ESA/390 PSW, which is as follows:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Zero</td>
</tr>
<tr>
<td>3</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>4</td>
<td>Wait state (W)</td>
</tr>
<tr>
<td>5</td>
<td>Problem state (P)</td>
</tr>
<tr>
<td>6</td>
<td>Address-space control (AS)</td>
</tr>
<tr>
<td>7</td>
<td>Condition code (CC)</td>
</tr>
<tr>
<td>8</td>
<td>Program mask</td>
</tr>
<tr>
<td>9</td>
<td>Zero</td>
</tr>
<tr>
<td>10</td>
<td>Extended addressing mode (EA)</td>
</tr>
<tr>
<td>11</td>
<td>Basic addressing mode (BA)</td>
</tr>
<tr>
<td>12</td>
<td>Zero</td>
</tr>
<tr>
<td>13</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>14</td>
<td>Wait state (W)</td>
</tr>
<tr>
<td>15</td>
<td>Problem state (P)</td>
</tr>
<tr>
<td>16-23</td>
<td>Address-space control (AS)</td>
</tr>
<tr>
<td>24-30</td>
<td>Program mask</td>
</tr>
<tr>
<td>31-63</td>
<td>Instruction address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16-23</td>
<td>Instruction address</td>
</tr>
</tbody>
</table>

General Registers 0-15: Contents of general registers 0-15 are stored in byte positions 32-159 as described in "General-Registers Control (R)" on page 10-134. When bits 32-63 or 0-63 of the general registers are stored, they are stored in ascending order of register numbers, starting with register 0 and continuing up to and including register 15.

Bytes 160-255 always remain unchanged. Bytes 168-255 are reserved. Bytes 160-167 are available for use by programming.

Special Conditions

The CPU must be in the primary-space mode or access-register mode, and bit 31 in bytes 44-47 of the dispatchable-unit control table must be one; otherwise, a special-operation exception is recognized. A special-operation exception is also recognized if the PSW control, bit 13 of bytes 0-3 of the trap control block is zero and bit 31 of the current PSW, the extended-addressing-mode bit, is one.

All protection mechanisms apply in the usual way to the accesses to the trap control block and trap save area. Access exceptions may or may not be recognized for sections of the trap control block and trap save area that are not referenced by the TRAP instruction.

The trap-program address in the trap control block is not tested before it replaces the instruction address in the PSW. An odd address will cause a specification exception to be recognized as part of the execution of the next instruction.

The operation is suppressed on all addressing and protection exceptions.

The priority of recognition of program exceptions for the instruction is shown in Figure 10-38 on page 10-137.

Condition Code: The code remains unchanged.
Program Exceptions:

- Access (fetch, trap control block; store, trap save area)
- Addressing (dispatchable-unit control table)
- Special operation
- Trace

Programming Notes:

1. It is intended that TRAP instructions will overlay instructions in an application program in order to give control to a trap program, which might be a program for performing fix-ups of data used by the application program, such as dates that may be a “Year-2000” problem. TRAP2 can overlay a two-byte instruction, and TRAP4 can overlay a four-byte instructions or the first four bytes of a six-byte instruction. The trap program is to simulate the overlaid instruction and perform fix-ups as appropriate, and it is then to return control to the application program.

2. The trap program can use the RESUME PROGRAM instruction to return control to the application program. For example, the trap program can restore the contents of all registers except access and general registers 15, and then, using those registers (or at least the general register) to address the trap save area, can restore the contents of those registers and also PSW fields from the trap save area. RESUME PROGRAM has control bits in its parameter list that allow it to restore PSW fields from a field having either the ESA/390 PSW format or the z/Architecture PSW format and to restore either bits 32-63 or 0-63 of a general register.

3. The trap control block and trap save area are in the home address space, and the trap program is in the primary address space. The trap-control-block address placed in general register 15 by TRAP can be useful to the trap program if (1) the primary address space and home address space are the same address space, (2) the trap control block and trap save area are at the same locations in the primary address space as in the home address space, or (3) the trap program can use access registers to access the home address space.

4. The storage-operand references for TRAP may be multiple-access references. (See “Storage-Operand Consistency” on page 5-98.)

1.-6. Exceptions with the same priority as the priority of program-interruption conditions for the general case.

7.A Access exceptions for second instruction halfword (TRAP4 only).

7.B Special-operation exception due to the CPU not being in the primary-space mode or access-register mode.

7.C.1 Addressing exception for access to dispatchable-unit control table.

7.C.2 Special-operation exception due to bit 31 in bytes 44-47 of dispatchable-unit control table being zero.

8.A Trace exceptions.

8.B.1 Access exceptions (fetch) for trap control block.

8.B.2 Special-operation exception due to PSW control in trap control block being zero and PSW bit 31 being one.

8.B.3 Access exceptions (store) for trap save area.

Figure 10-38. Priority of Execution: TRAP
### Chapter 11. Machine-Check Handling

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine-Check Detection</td>
<td>11-2</td>
</tr>
<tr>
<td>Correction of Machine Malfunctions</td>
<td>11-2</td>
</tr>
<tr>
<td>Error Checking and Correction</td>
<td>11-2</td>
</tr>
<tr>
<td>CPU Retry</td>
<td>11-2</td>
</tr>
<tr>
<td>Effects of CPU Retry</td>
<td>11-3</td>
</tr>
<tr>
<td>Checkpoint Synchronization</td>
<td>11-3</td>
</tr>
<tr>
<td>Handling of Machine Checks during Checkpoint Synchronization</td>
<td>11-3</td>
</tr>
<tr>
<td>Checkpoint Synchronization Operations</td>
<td>11-3</td>
</tr>
<tr>
<td>Checkpoint-Synchronization Action</td>
<td>11-4</td>
</tr>
<tr>
<td>Channel-Subsystem Recovery</td>
<td>11-4</td>
</tr>
<tr>
<td>Unit Deletion</td>
<td>11-4</td>
</tr>
<tr>
<td>Handling of Machine Checks</td>
<td>11-5</td>
</tr>
<tr>
<td>Validation</td>
<td>11-5</td>
</tr>
<tr>
<td>Invalid CBC in Storage</td>
<td>11-6</td>
</tr>
<tr>
<td>Programmed Validation of Storage</td>
<td>11-7</td>
</tr>
<tr>
<td>Invalid CBC in Storage Keys</td>
<td>11-7</td>
</tr>
<tr>
<td>Invalid CBC in Registers</td>
<td>11-10</td>
</tr>
<tr>
<td>Check-Stop State</td>
<td>11-11</td>
</tr>
<tr>
<td>System Check Stop</td>
<td>11-11</td>
</tr>
<tr>
<td>Machine-Check Interruption</td>
<td>11-11</td>
</tr>
<tr>
<td>Exigent Conditions</td>
<td>11-11</td>
</tr>
<tr>
<td>Repressible Conditions</td>
<td>11-12</td>
</tr>
<tr>
<td>Interruption Action</td>
<td>11-12</td>
</tr>
<tr>
<td>Point of Interruption</td>
<td>11-14</td>
</tr>
<tr>
<td>Machine-Check-Interruption Code</td>
<td>11-15</td>
</tr>
<tr>
<td>Subclass</td>
<td>11-16</td>
</tr>
<tr>
<td>System Damage</td>
<td>11-16</td>
</tr>
<tr>
<td>Instruction-Processing Damage</td>
<td>11-16</td>
</tr>
<tr>
<td>System Recovery</td>
<td>11-16</td>
</tr>
<tr>
<td>Timing-Facility Damage</td>
<td>11-16</td>
</tr>
<tr>
<td>External Damage</td>
<td>11-17</td>
</tr>
<tr>
<td>Degradation</td>
<td>11-17</td>
</tr>
<tr>
<td>Warning</td>
<td>11-17</td>
</tr>
<tr>
<td>Channel Report Pending</td>
<td>11-17</td>
</tr>
<tr>
<td>Service-Processor Damage</td>
<td>11-18</td>
</tr>
<tr>
<td>Channel-Subsystem Damage</td>
<td>11-18</td>
</tr>
<tr>
<td>Subclass Modifiers</td>
<td>11-18</td>
</tr>
<tr>
<td>Backed Up</td>
<td>11-18</td>
</tr>
<tr>
<td>Delayed Access Exception</td>
<td>11-18</td>
</tr>
<tr>
<td>Ancillary Report</td>
<td>11-18</td>
</tr>
<tr>
<td>Synchronous</td>
<td></td>
</tr>
<tr>
<td>Machine-Check-Interruption Conditions</td>
<td>11-18</td>
</tr>
<tr>
<td>Processing Backup</td>
<td>11-18</td>
</tr>
<tr>
<td>Processing Damage</td>
<td>11-19</td>
</tr>
<tr>
<td>Storage Errors</td>
<td>11-19</td>
</tr>
<tr>
<td>Storage Error Uncorrected</td>
<td>11-19</td>
</tr>
<tr>
<td>Storage Error Corrected</td>
<td>11-20</td>
</tr>
<tr>
<td>Storage-Key Error Uncorrected</td>
<td>11-20</td>
</tr>
<tr>
<td>Storage Degradation</td>
<td>11-20</td>
</tr>
<tr>
<td>Indirect Storage Error</td>
<td>11-20</td>
</tr>
<tr>
<td>Machine-Check Interruption-Code Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>PSW-MWP Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>PSW Mask and Key Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>PSW Program-Mask and Condition-Code Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>Failing-Storage-Address Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>External-Damage-Code Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>Floating-Point-Register Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>General-Register Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>Control-Register Validity</td>
<td>11-21</td>
</tr>
<tr>
<td>Storage Logical Validity</td>
<td>11-22</td>
</tr>
<tr>
<td>Access-Register Validity</td>
<td>11-22</td>
</tr>
<tr>
<td>TOD-Programmable-Register Validity</td>
<td>11-22</td>
</tr>
<tr>
<td>Floating-Point-Control-Register Validity</td>
<td>11-22</td>
</tr>
<tr>
<td>Clock-Comparator Validity</td>
<td>11-22</td>
</tr>
<tr>
<td>Machine-Check-Extended Interruption Information</td>
<td>11-22</td>
</tr>
<tr>
<td>Register-Save Areas</td>
<td>11-22</td>
</tr>
<tr>
<td>External-Damage Code</td>
<td>11-23</td>
</tr>
<tr>
<td>Failing-Storage Address</td>
<td>11-23</td>
</tr>
<tr>
<td>Handling of Machine-Check Conditions</td>
<td>11-23</td>
</tr>
<tr>
<td>Floating Interruption Conditions</td>
<td>11-23</td>
</tr>
<tr>
<td>Floating Machine-Check-Interruption Conditions</td>
<td>11-24</td>
</tr>
<tr>
<td>Floating I/O Interruptions</td>
<td>11-24</td>
</tr>
<tr>
<td>Machine-Check Masking</td>
<td>11-24</td>
</tr>
<tr>
<td>Channel-Report-Pending Subclass Mask</td>
<td>11-24</td>
</tr>
<tr>
<td>Recovery Subclass Mask</td>
<td>11-25</td>
</tr>
<tr>
<td>Degradation Subclass Mask</td>
<td>11-25</td>
</tr>
<tr>
<td>External-Damage Subclass Mask</td>
<td>11-25</td>
</tr>
<tr>
<td>Warning Subclass Mask</td>
<td>11-25</td>
</tr>
<tr>
<td>Machine-Check LogOut</td>
<td>11-25</td>
</tr>
<tr>
<td>Summary of Machine-Check Masking</td>
<td>11-25</td>
</tr>
</tbody>
</table>

The machine-check-handling mechanism provides extensive equipment-malfunction detection to ensure the integrity of system operation and to permit automatic recovery from some malfunctions. Equipment malfunctions and certain external disturbances are reported by means of a
machine-check interruption to assist in program-damage assessment and recovery. The interruption supplies the program with information about the extent of the damage and the location and nature of the cause. Equipment malfunctions, errors, and other situations which can cause machine-check interruptions are referred to as machine checks.

Machine-Check Detection

Machine-check-detection mechanisms may take many forms, especially in control functions for arithmetic and logical processing, addressing, sequencing, and execution. For program-addressable information, detection is normally accomplished by encoding redundancy into the information in such a manner that most failures in the retention or transmission of the information result in an invalid code. The encoding normally takes the form of one or more redundant bits, called check bits, appended to a group of data bits. Such a group of data bits and the associated check bits are called a checking block. The size of the checking block depends on the model.

The inclusion of a single check bit in the checking block allows the detection of any single-bit failure within the checking block. In this arrangement, the check bit is sometimes referred to as a “parity bit.” In other arrangements, a group of check bits is included to permit detection of multiple errors, to permit error correction, or both.

For checking purposes, the contents of the entire checking block, including the redundancy, are called the checking-block code (CBC). When a CBC completely meets the checking requirements (that is, no failure is detected), it is said to be valid. When both detection and correction are provided and a CBC is not valid but satisfies the checking requirements for correction (the failure is correctable), it is said to be near-valid. When a CBC does not satisfy the checking requirements (the failure is uncorrectable), it is said to be invalid.

Correction of Machine Malfunctions

Four mechanisms may be used to provide recovery from machine-detected malfunctions: error checking and correction, CPU retry, channel-subsystem recovery, and unit deletion.

Machine failures which are corrected successfully may or may not be reported as machine-check interruptions. If reported, they are system-recovery conditions, which permit the program to note the cause of CPU delay and to keep a log of such incidents.

Error Checking and Correction

When sufficient redundancy is included in circuitry or in a checking block, failures can be corrected. For example, circuitry can be triplicated, with a voting circuit to determine the correct value by selecting two matching results out of three, thus correcting a single failure. An arrangement for correction of failures of one order and for detection of failures of a higher order is called error checking and correction (ECC). Commonly, ECC allows correction of single-bit failures and detection of double-bit failures.

Depending on the model and the portion of the machine in which ECC is applied, correction may be reported as system recovery, or no report may be given.

Uncorrected errors in storage and in the storage key may be reported, along with a failing-storage address, to indicate where the error occurred. Depending on the situation, these errors may be reported along with system recovery or with the damage or backup condition resulting from the error.

CPU Retry

In some models, information about some portion of the state of the machine is saved periodically. The point in the processing at which this information is saved is called a checkpoint. The information saved is referred to as the checkpoint information. The action of saving the information is referred to as establishing a checkpoint. The action of discarding previously saved information is called invalidation of the checkpoint information.
The length of the interval between establishing checkpoints is model-dependent. Checkpoints may be established at the beginning of each instruction or several times within a single instruction, or checkpoints may be established less frequently.

Subsequently, this saved information may be used to restore the machine to the state that existed at the time when the checkpoint was established. After restoring the appropriate portion of the machine state, processing continues from the checkpoint. The process of restoring to a checkpoint and then continuing is called CPU retry.

CPU retry may be used for machine-check recovery, to effect nullification and suppression of instruction execution when certain program interruptions occur, and in other model-dependent situations.

Effects of CPU Retry
CPU retry is, in general, performed so that there is no effect on the program. However, change bits which have been changed from zeros to ones are not necessarily set back to zeros. As a result, change bits may appear to be set to ones for blocks which would have been accessed if restoring to the checkpoint had not occurred. If the path taken by the program is dependent on information that may be changed by another CPU or by a channel program or if an interruption occurs, then the final path taken by the program may be different from the earlier path; therefore, change bits may be ones because of stores along a path apparently never taken.

Checkpoint Synchronization
Checkpoint synchronization consists in the following steps.

1. The CPU operation is delayed until all conceptually previous accesses by this CPU to storage have been completed, both for purposes of machine-check detection and as observed by other CPUs and by channel programs.
2. All previous checkpoints, if any, are canceled.
3. Optionally, a new checkpoint is established.

The CPU operation is delayed until all of these actions appear to be completed, as observed by other CPUs and by channel programs.

Handling of Machine Checks during Checkpoint Synchronization
When, in the process of completing all previous stores as part of the checkpoint-synchronization action, the machine is unable to complete all stores successfully but can successfully restore the machine to a previous checkpoint, processing backup is reported.

When, in the process of completing all stores as part of the checkpoint-synchronization action, the machine is unable to complete all stores successfully and cannot successfully restore the machine to a previous checkpoint, the type of machine-check-interruption condition reported depends on the origin of the store. Failure to successfully complete stores associated with instruction execution may be reported as instruction-processing damage, or some less critical machine-check-interruption condition may be reported with the storage-logical-validity bit set to zero. A failure to successfully complete stores associated with the execution of an interruption, other than program or supervisor call, is reported as system damage.

When the machine check occurs as part of a checkpoint-synchronization action before the execution of an instruction, the execution of the instruction is nullified. When it occurs before the execution of an interruption, the interruption condition, if the interruption is external, I/O, or restart, is held pending. If the checkpoint-synchronization operation was a machine-check interruption, then along with the originating condition, either the storage-logical-validity bit is set to zero or instruction-processing damage is also reported. Program interruptions, if any, are lost.

Checkpoint-Synchronization Operations
All interruptions and the execution of certain instructions cause a checkpoint-synchronization action to be performed. The operations which cause a checkpoint-synchronization action are called checkpoint-synchronization operations and include:

- CPU reset
- All interruptions: external, I/O, machine check, program, restart, and supervisor call
- The BRANCH ON CONDITION (BCR) instruction with the M1 and R2 fields containing all ones and all zeros, respectively
• The instructions LOAD PSW, LOAD PSW EXTENDED, SET STORAGE KEY EXTENDED, and SUPERVISOR CALL
• All I/O instructions
• The instructions MOVE TO PRIMARY, MOVE TO SECONDARY, PROGRAM CALL, PROGRAM TRANSFER, SET ADDRESS SPACE CONTROL, and SET SECONDARY ASN, and PROGRAM RETURN when the state entry to be unstacked is a program-call state entry
• The four trace functions: branch tracing, ASN tracing, mode tracing, and explicit tracing
• PAGE IN and PAGE OUT

Programming Note: The instructions which are defined to cause the checkpoint-synchronization action invalidate checkpoint information but do not necessarily establish a new checkpoint. Additionally, the CPU may establish a checkpoint between any two instructions or units of operation, or within a single unit of operation. Thus, the point of interruption for the machine check is not necessarily at an instruction defined to cause a checkpoint-synchronization action.

Checkpoint-Synchronization Action
For all interruptions except I/O interruptions, a checkpoint-synchronization action is performed at the completion of the interruption. For I/O interruptions, a checkpoint-synchronization action may or may not be performed at the completion of the interruption. For all interruptions except program, supervisor-call, and exigent machine-check interruptions, a checkpoint-synchronization action is also performed before the interruption. The fetch access to the new PSW may be performed either before or after the first checkpoint-synchronization action. The store accesses and the changing of the current PSW associated with the interruption are performed after the first checkpoint-synchronization action and before the second.

For all checkpoint-synchronization instructions except BRANCH ON CONDITION (BCR), I/O instructions, and SUPERVISOR CALL, checkpoint-synchronization actions are performed before and after the execution of the instruction. For BCR, only one checkpoint-synchronization action is necessarily performed, and it may be performed either before or after the instruction address is updated. For SUPERVISOR CALL, a checkpoint-synchronization action is performed before the instruction is executed, including the updating of the instruction address in the PSW. The checkpoint-synchronization action taken after the supervisor-call interruption is considered to be part of the interruption action and not part of the instruction execution. For I/O instructions, a checkpoint-synchronization action is always performed before the instruction is executed and may or may not be performed after the instruction is executed.

The four trace functions — branch tracing, ASN tracing, mode tracing, and explicit tracing — cause checkpoint-synchronization actions to be performed before the trace action and after completion of the trace action.

Channel-Subsystem Recovery
When errors are detected in the channel subsystem, the channel subsystem attempts to analyze and recover the internal state associated with the various channel-subsystem functions and the state of the channel subsystem and various subchannels. This process, which is called channel-subsystem recovery, may result in a complete recovery or may result in the termination of one or more I/O operations and the clearing of the affected subchannels. Special channel-report-pending machine-check-interruption conditions may be generated to indicate to the program the status of the channel-subsystem recovery.

Malfunctions associated with the I/O operations, depending on the severity of the malfunction, may be reported by means of the I/O-interruption mechanism or by means of the channel-report-pending and channel-subsystem-damage machine-check-interruption conditions.

Unit Deletion
In some models, malfunctions in certain units of the system can be circumvented by discontinuing the use of the unit. Examples of cases where unit deletion may occur include the disabling of all or a portion of a cache or of a translation-lookaside buffer (TLB). Unit deletion may be reported as a degradation machine-check-interruption condition.
Handling of Machine Checks

A machine check is caused by a machine malfunction and not by data or instructions. This is ensured during the power-on sequence by initializing the machine controls to a valid state and by placing valid CBC in the CPU registers, in the storage keys, and in main storage.

Designation of an unavailable component, such as a storage location, subchannel, or I/O device, does not cause a machine-check indication. Instead, such a condition is indicated by the appropriate program or I/O interruption or condition-code setting. In particular, an attempt to access a storage location which is not in the configuration, or which has power off at the storage unit, results in an addressing exception when detected by the CPU and does not generate a machine-check condition, even though the storage location or its associated storage key has invalid CBC. Similarly, if the channel subsystem attempts to access such a location, an I/O-interruption condition indicating program check is generated rather than a machine-check condition.

A machine check is indicated whenever the result of an operation could be affected by information with invalid CBC or when any other malfunction makes it impossible to establish reliably that an operation can be, or has been, performed correctly. When information with invalid CBC is fetched but not used, the condition may or may not be indicated, and the invalid CBC is preserved.

When a machine malfunction is detected, the action taken depends on the model, the nature of the malfunction, and the situation in which the malfunction occurs. Malfunctions affecting operator-facility actions may result in machine checks or may be indicated to the operator. Malfunctions affecting certain other operations such as SIGNAL PROCESSOR may be indicated by means of a condition code or may result in a machine-check-interruption condition.

A malfunction detected as part of an I/O operation may cause a machine-check-interruption condition, an I/O-error condition, or both. I/O-error conditions are indicated by an I/O interruption or by the appropriate condition-code setting during the execution of an I/O instruction. When the machine reports a failing-storage location detected during an I/O operation, both I/O-error and machine-check conditions may be indicated. The I/O-error condition is the primary indication to the program. The machine-check condition is a secondary indication, which is presented as system recovery together with a failing-storage address.

Certain malfunctions detected as part of I/O instructions and I/O operations are reported by means of special machine-check conditions called I/O machine-check conditions. Thus, malfunctions detected as part of an operation which is I/O related may be reported, depending on the error, in any of three ways: I/O-error condition, I/O machine-check condition, or non-I/O machine-check condition. In some cases, the definition requires the error to be reported by only one of these mechanisms; in other cases, any one, or in some cases, more than one, may be indicated.

Programming Note: Although the definition for machine-check conditions is that they are caused by machine malfunctions and not by data and instructions, there are certain unusual situations in which machine-check conditions are caused by events which are not machine malfunctions. Two examples follow:

1. In some cases, the channel-report-pending machine-check-interruption condition indicates a non-error situation. For example, this condition is generated at the completion of the function specified by RESET CHANNEL PATH.

2. Improper use of DIAGNOSE may result in machine-check conditions.

Validation

Machine errors can be generally classified as solid or intermittent, according to the persistence of the malfunction. A persistent machine error is said to be solid, and one that is not persistent is said to be intermittent. In the case of a register or storage location, a third type of error must be considered, called externally generated. An externally generated error is one where no failure exists in the register or storage location but invalid CBC has been introduced into the location by actions external to the location. For example, the value could be affected by a power transient, or an incorrect value may have been introduced when the information was placed at the location.
Invalid CBC is preserved as invalid when information with invalid CBC is fetched or when an attempt is made to update only a portion of the checking block. When an attempt is made to replace the contents of the entire checking block and the block contains invalid CBC, it depends on the operation and the model whether the block remains with invalid CBC or is replaced. An operation which replaces the contents of a checking block with valid CBC, while ignoring the current contents, is called a validation operation. Validation is used to place a valid CBC in a register or at a location which has an intermittent or externally generated error.

Validating a checking block does not ensure that a valid CBC will be observed the next time the checking block is accessed. If the failure is solid, validation is effective only if the information placed in the checking block is such that the failing bits are set to the value to which they fail. If an attempt is made to set the bits to the state opposite to that in which they fail, then the validation will not be effective. Thus, for a solid failure, validation is only useful to eliminate the error condition, even though the underlying failure remains, thereby reducing the exposure to additional reports. The locations, however, cannot be used, since invalid CBC will result from attempts to store other values at the location. For an intermittent failure, however, validation is useful to restore a valid CBC such that a subsequent partial store into the checking block will be permitted. (A partial store is a store into a checking block without replacing the entire checking block.)

When a checking block consists of multiple bytes in storage, or multiple bits in CPU registers, the invalid CBC can be made valid only when all of the bytes or bits are replaced simultaneously.

For each type of field in the system, certain instructions are defined to validate the field. Depending on the model, additional instructions may also perform validation; or, in some models, a register is automatically validated as part of the machine-check-interruption sequence after the original contents of the register are placed in the appropriate save area.

When an error occurs in a checking block, the original information contained in the checking block should be considered lost even after validation. Automatic register validation leaves the contents unpredictable. Programmed and manual validation of checking blocks causes the contents to be changed explicitly.

**Programming Note:** The machine-check-interruption handler must assume that the registers require validation. Thus, each register should be loaded, using an instruction defined to validate, before the register is used or stored.

### Invalid CBC in Storage

The size of the checking block in storage depends on the model but is never more than 4K bytes.

When invalid CBC is detected in storage, a machine-check condition may occur; depending on the circumstances, the machine-check condition may be system damage, instruction-processing damage, or system recovery. If the invalid CBC is detected as part of the execution of a channel program, the error is reported as an I/O-error condition. When a CCW, indirect-data-address word, or data is prefetched from storage, is found to have invalid CBC, but is not used in the channel program, the condition is normally not reported as an I/O-error condition. The condition may or may not be reported as a machine-check-interruption condition. Invalid CBC detected during accesses to storage for other than CPU-related accesses may be reported as system recovery with storage error uncorrected indicated, since the primary error indication is reported by some other means.

When the storage checking block consists of multiple bytes and contains invalid CBC, special storage-validation procedures are generally necessary to restore or place new information in the checking block. Validation of storage is provided with the manual load-clear and system-reset-clear operations and is also provided as a program function. Programmed storage validation is done a block at a time, by executing the privileged instruction TEST BLOCK. Manual storage validation by clear reset validates all blocks which are available in the configuration.

A checking block with invalid CBC is never validated unless the entire contents of the checking block are replaced. An attempt to store into a checking block having invalid CBC, without replacing the entire checking block, leaves the data in the checking block (including the check bits) unchanged. Even when an instruction or a
channel-program-input operation specifies that the entire contents of a checking block are to be replaced, validation may or may not occur, depending on the operation and the model.

**Programming Note:** Machine-check conditions may be reported for prefetched and unused data. Depending on the model, such situations may or may not be successfully retried. For example, a BRANCH AND LINK (BALR) instruction which specifies an R₀ field of zero will never branch, but on some models a prefetch of the location designated by register 0 may occur. Access exceptions associated with this prefetch will not be reported. However, if an invalid checking-block code is detected, CPU retry may be attempted. Depending on the model, the prefetch may recur as part of the retry, and thus the retry will not be successful. Even when the CPU retry is successful, the performance degradation of such a retry is significant, and system recovery may be presented, normally with a failing-storage address. To avoid continued degradation, the program should initiate proceedings to eliminate use of the location and to validate the location.

**Programmed Validation of Storage**

Provided that an invalid CBC does not exist in the storage key associated with a 4K-byte block, the instruction TEST BLOCK causes the entire 4K-byte block to be set to zeros with a valid CBC, regardless of the current contents of the storage. TEST BLOCK thus removes an invalid CBC from a location in storage which has an intermittent, or one-time, failure. However, if a permanent failure exists in a portion of the storage, a subsequent fetch may find an invalid CBC.

**Invalid CBC in Storage Keys**

Depending on the model, each storage key may be contained in a single checking block, or the access-control and fetch-protection bits and the reference and change bits may be in separate checking blocks. The action taken when the storage key has invalid CBC is described in Figure 11-1 on page 11-8. The figure indicates the action taken for the case when the access-control and fetch-protection bits are in one checking block and the reference and change bits are in a separate checking block. In machines where both fields are included in a single checking block, the action taken is the combination of the actions for each field in error, except that completion is permitted only if an error in all affected fields permits completion. References to main storage to which key-controlled protection does not apply are treated as if an access key of zero is used for the reference. This includes such references as channel-program references during initial program loading and implicit references, such as interruption action and DAT-table accesses.
<table>
<thead>
<tr>
<th>Type of Reference</th>
<th>Action Taken on Invalid CBC</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For Access-Control and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch-Protection Bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For Reference and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Change Bits</td>
<td></td>
</tr>
<tr>
<td>SET STORAGE KEY</td>
<td>Complete; validate.</td>
<td></td>
</tr>
<tr>
<td>EXTENDED</td>
<td>Complete; validate.</td>
<td></td>
</tr>
<tr>
<td>INSERT STORAGE KEY</td>
<td>PD; preserve.</td>
<td></td>
</tr>
<tr>
<td>EXTENDED</td>
<td>PD; preserve.</td>
<td></td>
</tr>
<tr>
<td>RESET REFERENCE</td>
<td>PD or complete; preserve.</td>
<td></td>
</tr>
<tr>
<td>BIT EXTENDED</td>
<td>PD; preserve.</td>
<td></td>
</tr>
<tr>
<td>INSERT VIRTUAL</td>
<td>PD; preserve.</td>
<td></td>
</tr>
<tr>
<td>STORAGE KEY or</td>
<td>CPF; preserve.</td>
<td></td>
</tr>
<tr>
<td>TEST PROTECTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU prefetch</td>
<td>CPF; preserve.</td>
<td></td>
</tr>
<tr>
<td>(information not</td>
<td></td>
<td></td>
</tr>
<tr>
<td>used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Channel-program</td>
<td>IPF; preserve.</td>
<td></td>
</tr>
<tr>
<td>pre-fetch (infor-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mation not used)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch, nonzero</td>
<td>MC; preserve.</td>
<td></td>
</tr>
<tr>
<td>access key</td>
<td>MC or complete; preserve.</td>
<td></td>
</tr>
<tr>
<td>Store(^1),</td>
<td>MC(^2); preserve.</td>
<td></td>
</tr>
<tr>
<td>nonzero access</td>
<td>MC and preserve; or</td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>complete(^3) and correct.</td>
<td></td>
</tr>
<tr>
<td>Fetch, zero</td>
<td>MC or complete; preserve.</td>
<td></td>
</tr>
<tr>
<td>access key(^4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store(^1), zero</td>
<td>MC and preserve; or</td>
<td></td>
</tr>
<tr>
<td>access key(^2)</td>
<td>complete(^3) and correct.</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

1. CPU virtual- and logical-address store accesses are subject to page protection. When the page-protection bit is one, the location will not be changed; however, the machine may indicate a machine-check condition if the storage key or the data itself has invalid CBC.

2. The contents of the main-storage location are not changed.

3. The contents of the reference and change bits are set to ones if the "complete" action is taken.

4. The action shown for an access key of zero is also applicable to references to which key-controlled protection does not apply.

*Figure 11-1 (Part 1 of 2). Invalid CBC in Storage Keys*
**Explanation (Continued):**

**Complete** The condition does not cause termination of the execution of the instruction, and, unless an unrelated condition prohibits it, the execution of the instruction is completed, ignoring the error condition. No machine-check-damage conditions are reported, but system recovery may be reported.

**Correct** The reference and change bits are set to ones with valid CBC.

**Preserve** The contents of the entire checking block having invalid CBC are left unchanged.

**Validate** The entire key is set to the new value with valid CBC.

**CPF** Invalid CBC in the storage key for a CPU prefetch which is unused, or for instructions which do not examine the reference and change bits, may result in any of the following situations:
- The operation is completed; no machine-check condition is reported.
- The operation is completed; system recovery, with storage-key error uncorrected, is reported.
- Instruction-processing damage, with or without backup and with storage-key error uncorrected, is reported.

**IPF** Invalid CBC in the storage key for a channel-program prefetch which is unused may result in any of the following:
- The I/O operation is completed; no machine-check condition is reported.
- The I/O operation is completed; system recovery, with storage-key error uncorrected, is reported.

**MC** Same as PD for CPU references, but a channel-subsystem reference may result in the following combinations of I/O-error conditions and machine-check conditions:
- An I/O-error condition is reported; no machine-check condition is reported.
- An I/O-error condition is reported; system recovery, with or without storage-key error uncorrected, is reported.

**PD** Instruction-processing damage, with or without backup and with or without storage-key error uncorrected, is reported.

**Note:** When storage-key error uncorrected is reported, a failing-storage address may or may not also be reported.

*Figure 11-1 (Part 2 of 2). Invalid CBC in Storage Keys*
Invalid CBC in Registers

When invalid CBC is detected in a CPU register, a machine-check condition may be recognized. CPU registers include the general, floating-point, floating-point-control, access, control, and TOD programmable registers, the current PSW, the prefix register, the TOD clock, the CPU timer, and the clock comparator.

When a machine-check interruption occurs, whether or not it is due to invalid CBC in a CPU register, the following actions affecting the CPU registers, other than the prefix register and the TOD clock, are taken as part of the interruption.

1. The contents of the registers are saved in assigned storage locations. Any register which is in error is identified by a corresponding validity bit of zero in the machine-check-interruption code. Malfunctions detected during register saving do not result in additional machine-check-interruption conditions; instead, the correctness of all the information stored is indicated by the appropriate setting of the validity bits.

2. On some models, registers with invalid CBC are then validated, their actual contents being unpredictable. On other models, programmed validation is required.

The prefix register and the TOD clock are not stored during a machine-check interruption, have no corresponding validity bit, and are not validated.

On those models in which registers are not automatically validated as part of the machine-check interruption, a register with invalid CBC will not cause a machine-check-interruption condition unless the contents of the register are actually used. In these models, each register may consist of one or more checking blocks, but multiple registers are not included in a single checking block. When only a portion of a register is accessed, invalid CBC in the unused portion of the same register may cause a machine-check-interruption condition. For example, invalid CBC in the right half of a floating-point register may cause a machine-check-interruption condition if a LOAD (LE) operation attempts to replace the left half, or short form, of the register.

Invalid CBC associated with the prefix register cannot safely be reported by the machine-check interruption, since the interruption itself requires that the prefix value be applied to convert real addresses to the corresponding absolute addresses. Invalid CBC in the prefix register causes the CPU to enter the check-stop state immediately.

On those models which do not validate registers during a machine-check interruption, the following instructions will cause validation of a register, provided the information in the register is not used before the register is validated. Other instructions, although they replace the entire contents of a register, do not necessarily cause validation.

General registers are validated by BRANCH AND SAVE and LOAD ADDRESS executed in the 64-bit addressing mode, by LOAD (LGR), and by LOAD (LG) and LOAD MULTIPLE (LMG) if the operand is on a doubleword boundary.

Floating-point registers are validated by LOAD (LDR) and, if the operand is on a doubleword boundary, by LOAD (LD).

The floating-point-control register is validated by LOAD FLOATING POINT CONTROL REGISTER.

Access registers are validated by LOAD ACCESS MULTIPLE. Only the even-odd access-register pairs that are included in the set of access registers specified for LOAD ACCESS MULTIPLE are validated. Thus, when a single access register is specified, or when a pair of access registers starting with an odd-numbered register is specified, no register is validated.

Control registers may be validated either singly or in groups by using the instruction LOAD CONTROL (LCTLG).

The TOD programmable register, CPU timer, clock comparator, and prefix register are validated by SET CLOCK PROGRAMMABLE REGISTER, SET CPU TIMER, SET CLOCK COMPARATOR, and SET PREFIX, respectively.

The TOD clock is validated by a SET CLOCK instruction that sets the clock.

Programming Note: Depending on the register and the model, the contents of a register may be validated by the machine-check interruption, or the
model may require that a program execute a validating instruction after the machine-check interruption has occurred. In the case of the CPU timer, depending on the model, both the machine-check interruption and validating instructions may be required to restore the CPU timer to full working order.

Check-Stop State

In certain situations, it is impossible or undesirable to continue operation when a machine error occurs. In these cases, the CPU may enter the check-stop state, which is indicated by the check-stop indicator.

In general, the CPU may enter the check-stop state whenever an uncorrectable error or other malfunction occurs and the machine is unable to recognize a specific machine-check-interruption condition.

The CPU always enters the check-stop state if any of the following conditions exists:

- PSW bit 13 is zero, and an exigent machine-check condition is generated.
- During the execution of an interruption due to one exigent machine-check condition, another exigent machine-check condition is detected.
- During a machine-check interruption, the machine-check-interruption code cannot be stored successfully, or the new PSW cannot be fetched successfully.
- Invalid CBC is detected in the prefix register.
- A malfunction in the receiving CPU, which is detected after accepting the order, prevents the successful completion of a SIGNAL PROCESSOR order and the order was a reset, or the receiving CPU cannot determine what the order was. The receiving CPU enters the check-stop state.

There may be many other conditions for particular models when an error may cause check stop.

When the CPU is in the check-stop state, instructions and interruptions are not executed. The TOD clock is normally not affected by the check-stop state. The CPU timer may or may not run in the check-stop state, depending on the error and the model. The start key and stop key are not effective in this state.

The CPU may be removed from the check-stop state by CPU reset.

In a multiprocessing configuration, a CPU entering the check-stop state generates a request for a malfunction-alert external interruption to all CPUs in the configuration. Except for the reception of a malfunction alert, other CPUs and the I/O system are normally unaffected by the check-stop state in a CPU. However, depending on the nature of the condition causing the check stop, other CPUs may also be delayed or stopped, and channel subsystem and I/O activity may be affected.

System Check Stop

In a multiprocessing configuration, some errors, malfunctions, and damage conditions are of such severity that the condition causes all CPUs in the configuration to enter the check-stop state. This condition is called a system check stop. The state of the channel subsystem and I/O activity is unpredictable.

Machine-Check Interruption

A request for a machine-check interruption, which is made pending as the result of a machine check, is called a machine-check-interruption condition. There are two types of machine-check-interruption conditions: exigent conditions and repressible conditions.

Exigent Conditions

Exigent machine-check-interruption conditions are those in which damage has or would have occurred such that execution of the current instruction or interruption sequence cannot safely continue. Exigent conditions include two subclasses: instruction-processing damage and system damage. In addition to indicating specific exigent conditions, system damage is used to report any malfunction or error which cannot be isolated to a less severe report.

Exigent conditions for instruction sequences can be either nullifying exigent conditions or terminating exigent conditions, according to whether the instructions affected are nullified or terminated. Exigent conditions for interruption sequences are terminating exigent conditions. The terms “nullification” and “termination” have the same
meanings as those used in Chapter 5, “Program Execution,” except that more than one instruction may be involved. Thus, a nullifying exigent condition indicates that the CPU has returned to the beginning of a unit of operation prior to the error. A terminating exigent condition means that the results of one or more instructions may have unpredictable values.

Repressible Conditions

Repressible machine-check-interruption conditions are those in which the results of the instruction-processing sequence have not been affected. Repressible conditions can be delayed, until the completion of the current instruction or even longer, without affecting the integrity of CPU operation. Repressible conditions are of three groups: recovery, alert, and repressible damage. Each group includes one or more subclasses.

A malfunction in the CPU, storage, or operator facilities which has been successfully corrected or circumvented internally without logical damage is called a recovery condition. Depending on the model and the type of malfunction, some or all recovery conditions may be discarded and not reported. Recovery conditions that are reported are grouped in one subclass, system recovery.

A machine-check-interruption condition not directly related to a machine malfunction is called an alert condition. The alert conditions are grouped in two subclasses: degradation and warning.

A malfunction resulting in an incorrect state of a portion of the system not directly affecting sequential CPU operation is called a repressible-damage condition. Repressible-damage conditions are grouped in five subclasses, according to the function affected: timing-facility damage, external damage, channel report pending, channel-subsystem damage, and service-processor damage.

Programming Notes:

1. Even though repressible conditions are usually reported only at normal points of interruption, they may also be reported with exigent machine-check conditions. Thus, if an exigent machine-check condition causes an instruction to be abnormally terminated and a machine-check interruption occurs to report the exigent condition, any pending repressible conditions may also be reported. The meaningfulness of the validity bits depends on what exigent condition is reported.

2. Classification of damage as either exigent or repressible does not imply the severity of the damage. The distinction is whether action must be taken as soon as the damage is detected (exigent) or whether the CPU can continue processing (repressible). For a repressible condition, the current instruction can be completed before taking the machine-check interruption if the CPU is enabled for machine checks; if the CPU is disabled for machine checks, the condition can safely be kept pending until the CPU is again enabled for machine checks.

For example, the CPU may be disabled for machine-check interruptions because it is handling an earlier instruction-processing-damage interruption. If, during that time, an I/O operation encounters a storage error, that condition can be kept pending because it is not expected to interfere with the current machine-check processing. If, however, the CPU also makes a reference to the area of storage containing the error before re-enabling machine-check interruptions, another instruction-processing-damage condition is created, which is treated as an exigent condition and causes the CPU to enter the check-stop state.

3. A repressible condition may be a floating condition. A floating repressible condition is eligible to cause an interruption on any CPU in the configuration. At the point when a CPU performs an interruption for a floating repressible condition, the condition is no longer eligible to cause an interruption on the remaining CPUs in the configuration.

Interruption Action

A machine-check interruption causes the following actions to be taken. An architectural-mode identification with the value 01 hex is stored at real location 163. The PSW reflecting the point of interruption is stored as the machine-check old PSW in the quadword at real location 352. The contents of other registers are stored in register-save areas at real locations 4608-4863, 4892-4895, 4900-4911, 4913-4919, and 4928-5119. After the contents of the registers are
stored in register-save areas, depending on the model, the registers may be validated with the contents being unpredictable. A machine-check-interruption code (MCIC) of eight bytes is stored at real locations 232-239. An external-damage code may be stored at real locations 244-247, and a failing-storage address may be stored at real locations 248-255. The new PSW is fetched from real locations 480-495. In addition, a machine-check logout may have occurred.

The machine-generated addresses used to access the old and new PSW, the MCIC, extended interruption information, and the fixed-logout area are all real addresses.

The fields in assigned storage locations that are accessed during the machine-check interruption are summarized in Figure 11-2.

<table>
<thead>
<tr>
<th>Information Stored (Fetched)</th>
<th>Starting Location</th>
<th>Length in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural-mode identification</td>
<td>163</td>
<td>1</td>
</tr>
<tr>
<td>Old PSW</td>
<td>352</td>
<td>16</td>
</tr>
<tr>
<td>New PSW (fetched)</td>
<td>480</td>
<td>16</td>
</tr>
<tr>
<td>Machine-check-interruption code</td>
<td>232</td>
<td>8</td>
</tr>
<tr>
<td>Register-save areas</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point registers 0-15</td>
<td>4608</td>
<td>128</td>
</tr>
<tr>
<td>General registers 0-15</td>
<td>4736</td>
<td>128</td>
</tr>
<tr>
<td>Floating-point control register</td>
<td>4892</td>
<td>4</td>
</tr>
<tr>
<td>TOD programmable register</td>
<td>4900</td>
<td>4</td>
</tr>
<tr>
<td>CPU timer</td>
<td>4904</td>
<td>8</td>
</tr>
<tr>
<td>Clock comparator</td>
<td>4913</td>
<td>7</td>
</tr>
<tr>
<td>Access registers 0-15</td>
<td>4926</td>
<td>64</td>
</tr>
<tr>
<td>Control registers 0-15</td>
<td>4992</td>
<td>128</td>
</tr>
<tr>
<td>Extended interruption information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External-damage code</td>
<td>244</td>
<td>4</td>
</tr>
<tr>
<td>Failing-storage address</td>
<td>248</td>
<td>8</td>
</tr>
<tr>
<td>Fixed-logout area</td>
<td>4864</td>
<td>16</td>
</tr>
</tbody>
</table>

Explanation:
* All locations are in real storage.

Figure 11-2. Machine-Check-Interruption Locations

If the machine-check-interruption code cannot be stored successfully or the new PSW cannot be fetched successfully, the CPU enters the check-stop state.

A repressible machine-check condition can initiate a machine-check interruption only if both PSW bit 13 is one and the associated subclass mask bit, if any, in control register 14 is also one. When it occurs, the interruption does not terminate the execution of the current instruction; the interruption is taken at a normal point of interruption, and no program or supervisor-call interruptions are eliminated. If the machine check occurs during the execution of a machine function, such as a CPU-timer update, the machine-check interruption takes place after the machine function has been completed.

When the CPU is disabled for a particular repressible machine-check condition, the condition remains pending. Depending on the model and the condition, multiple repressible conditions may be held pending for a particular subclass, or only one condition may be held pending for a particular subclass, regardless of the number of conditions that may have been detected for that subclass.

When a repressible machine-check interruption occurs because the interruption condition is in a subclass for which the CPU is enabled, pending conditions in other subclasses may also be indicated by the same interruption code, even though the CPU is disabled for those subclasses. All indicated conditions are then cleared.

If a machine check which is to be reported as a system-recovery condition is detected during the execution of the interruption procedure due to a previous machine-check condition, the system-recovery condition may be combined with the other conditions, discarded, or held pending.

An exigent machine-check condition can cause a machine-check interruption only when PSW bit 13 is one. When a nullifying exigent condition causes a machine-check interruption, the interruption is taken at a normal point of interruption. When a terminating exigent condition causes a machine-check interruption, the interruption terminates the execution of the current instruction and may eliminate the program and supervisor-call interruptions, if any, that would have occurred if execution had continued. Proper execution of the interruption sequence, including the storing of the old PSW and other information, depends on the nature of the malfunction. When an exigent machine-check condition occurs during the execution of a machine function, such as a CPU-timer update, the sequence is not necessarily completed.

If, during the execution of an interruption due to one exigent machine-check condition, another exigent machine check is detected, the CPU enters the check-stop state. If an exigent machine check is detected during an interruption due to a repressible machine-check condition, system damage is reported.
When PSW bit 13 is zero, an exigent machine-check condition causes the CPU to enter the check-stop state.

Machine-check-interruption conditions are handled in the same manner regardless of whether the wait-state bit in the PSW is one or zero: a machine-check condition causes an interruption if the CPU is enabled for that condition.

Machine checks which occur while the rate control is set to the instruction-step position are handled in the same manner as when the control is set to the process position; that is, recovery mechanisms are active, and machine-check interruptions occur when allowed. Machine checks occurring during a manual operation may be indicated to the operator, may generate a system-recovery condition, may result in system damage, or may cause a check stop, depending on the model.

Every reasonable attempt is made to limit the side effects of any machine check and the associated interruption. Normally, interruptions, as well as the progress of I/O operations, remain unaffected. The malfunction, however, may affect these activities, and, if the currently active PSW has bit 13 set to one, the machine-check interruption will indicate the total extent of the damage caused, and not just the damage which originated the condition.

**Point of Interruption**

The point in the processing which is indicated by the interruption and used as a reference point by the machine to determine and indicate the validity of the status stored is referred to as the point of interruption.

Because of the checkpoint capability in models with CPU retry, the interruption resulting from an exigent machine-check-interruption condition may indicate a point in the CPU processing sequence which is logically prior to the error. Additionally, the model may have some choice as to which point in the CPU processing sequence the interruption is indicated, and, in some cases, the status which can be indicated as valid depends on the point chosen.

Only certain points in the processing may be used as a point of interruption. For repressible machine-check interruptions, the point of interruption must be after one unit of operation is completed and any associated program or supervisor-call interruption is taken, and before the next unit of operation is begun.

Exigent machine-check conditions for instruction sequences are those in which damage has or would have occurred to the instruction stream. Thus, the damage can normally be associated with a point part way through an instruction, and this point is called the point of damage. In some cases, there may be one or more instructions separating the point of damage and the point of interruption, and the processing associated with one or more instructions may be damaged. When the point of interruption is a point prior to the point of damage due to a nullifiable exigent machine-check condition, the point of interruption can be only at the same points as for repressible machine-check conditions.

In addition to the point of interruption permitted for repressible machine-check conditions, the point of interruption for a terminating exigent machine-check condition may also be after the unit of operation is completed but before any associated program or supervisor-call interruption occurs. In this case, a valid PSW instruction address is defined as that which would have been stored in the old PSW for the program or supervisor-call interruption. Since the operation has been terminated, the values in the result fields, other than the instruction address, are unpredictable. Thus, the validity bits associated with fields which are due to be changed by the instruction stream are meaningless when a terminating exigent machine-check condition is reported.

When the point of interruption and the point of damage due to an exigent machine-check condition are separated by a checkpoint-synchronization function, the damage has not been isolated to a particular program, and system damage is indicated.

When an exigent machine-check-interruption condition occurs, the point of interruption which is chosen affects the amount of damage which must be indicated. An attempt is made, when possible, to choose a point of interruption which permits the minimum indication of damage. In general, the preference is the interruption point immediately preceding the error.
When all the status information stored as a result of an exigent machine-check-interruption condition does not reflect the same point, an attempt is made, when possible, to choose the point of interruption so that the instruction address which is stored in the machine-check old PSW is valid.

**Machine-Check-Interruption Code**

On all machine-check interruptions, a machine-check-interruption code (MCIC) is stored in the doubleword starting at real location 232. The code has the format shown in Figure 11-3.

Bits in the MCIC which are not assigned or not implemented by a particular model are stored as zeros.

---

**Figure 11-3. Machine-Check Interruption-Code Format**
Subclass

Bits 0-2 and 4-11 are the subclass bits which identify the type of machine-check condition causing the interruption. At least one of the subclass bits is stored as a one. When multiple errors have occurred, several subclass bits may be set to ones.

System Damage

Bit 0 (SD), when one, indicates that damage has occurred which cannot be isolated to one or more of the less severe machine-check subclasses. When system damage is indicated, the ancillary-report bit, bit 44, is meaningful, the remaining bits in the machine-check-interruption code are not meaningful, and information stored in the register-save areas and machine-check extended-interruption fields is not meaningful.

System damage is a terminating exigent condition and has no subclass-mask bit.

Instruction-Processing Damage

Bit 1 (PD), when one, indicates that damage has occurred to the instruction processing of the CPU.

The exact meaning of bit 1 depends on the setting of the backed-up bit, bit 14. When the backed-up bit is one, the condition is called processing backup. When the backed-up bit is zero, the condition is called processing damage. These two conditions are described in "Synchronous Machine-Check-Interruption Conditions" on page 11-18.

Instruction-processing damage can be a nullifying or a terminating exigent condition and has no subclass-mask bit.

System Recovery

Bit 2 (SR), when one, indicates that malfunctions were detected but did not result in damage or have been successfully corrected. Some malfunctions detected as part of an I/O operation may result in a system-recovery condition in addition to an I/O-error condition. The presence and extent of the system-recovery capability depend on the model.

System recovery is a repressible condition. It is masked by the recovery subclass-mask bit, which is in bit position 36 of control register 14.

Programming Notes:

1. System recovery may be used to report a failing-storage address detected by a CPU prefetch or by an I/O operation.

2. Unless the corresponding validity bits are ones, the indication of system recovery does not imply storage logical validity or that the fields stored as a result of the machine-check interruption are valid.

Timing-Facility Damage

Bit 4 (CD), when one, indicates that damage has occurred to the TOD clock, CPU timer, clock comparator, or TOD programmable register, or to the CPU-timer or clock-comparator external-interruption conditions. The timing-facility-damage machine-check condition is set whenever any of the following occurs:

1. The TOD clock enters the error or not-operational state.

2. The CPU timer is damaged, and the CPU is enabled for CPU-timer external interruptions. On some models, this condition may be recognized even when the CPU is not enabled for CPU-timer interruptions. Depending on the model, the machine-check condition may be generated only as the CPU timer enters an error state. Or, the machine-check condition may be continuously generated whenever the CPU is enabled for CPU-timer interruptions, until the CPU timer is validated.

3. The clock comparator is damaged, and the CPU is enabled for clock-comparator external interruptions. On some models, this condition may be recognized even when the CPU is not enabled for clock-comparator interruptions.

Timing-facility damage may also be set along with instruction-processing damage when an instruction which accesses the TOD clock, CPU timer, or clock comparator produces incorrect results. Depending on the model, the TOD programmable register, CPU timer, or clock comparator may be validated by the interruption which reports the TOD clock, CPU timer, or clock comparator as invalid.

Timing-facility damage is a repressible condition. It is masked by the external-damage subclass-mask bit, which is in bit position 38 of control register 14.
Timing-facility-damage conditions for the CPU timer and the clock comparator are not recognized on most models when these facilities are not in use. The facilities are considered not in use when the CPU is disabled for the corresponding external interruptions (PSW bit 7, or the subclass-mask bits, bits 52 and 53 of control register 0, are zeros), and when the corresponding set and store instructions are not executed. Timing-facility-damage conditions that are already pending remain pending, however, when the CPU is disabled for the corresponding external interruption.

Timing-facility-damage conditions due to damage to the TOD clock are always recognized.

**External Damage**

Bit 5 (ED), when one, indicates that damage has occurred during operations not directly associated with processing the current instruction.

When bit 5, external damage, is one and bit 26, external-damage-code validity, is also one, the external-damage code has been stored to indicate, in more detail, the cause of the external-damage machine-check interruption. When the external damage cannot be isolated to one or more of the conditions as defined in the external-damage code, or when the detailed indication for the condition is not implemented by the model, external damage is indicated with bit 26 set to zero. The presence and extent of reporting external damage depend on the model.

External damage is a repressible condition. It is masked by the external-damage subclass-mask bit, which is in bit position 38 of control register 14.

**Degradation**

Bit 7 (DG), when one, indicates that continuous degradation of system performance, more serious than that indicated by system recovery, has occurred. Degradation may be reported when system-recovery conditions exceed a machine-preestablished threshold or when unit deletion has occurred. The presence and extent of the degradation-report capability depend on the model.

Degradation is a repressible condition. It is masked by the degradation subclass-mask bit, which is in bit position 37 of control register 14.

**Warning**

Bit 8 (W), when one, indicates that damage is imminent in some part of the system (for example, that power is about to fail, or that a loss of cooling is occurring). Whether warning conditions are recognized depends on the model.

If the condition responsible for the imminent damage is removed before the interruption request is honored (for example, if power is restored), the request does not remain pending, and no interruption occurs. Conversely, the request is not cleared by the interruption, and, if the condition persists, more than one interruption may result from the same condition.

Warning is a repressible condition. It is masked by the warning subclass-mask bit, which is in bit position 39 of control register 14.

**Channel Report Pending**

Bit 9 (CP), when one, indicates that a channel report, consisting of one or more channel-report words, has been made pending, and the contents of the channel-report words describe, in further detail, the effect of the malfunction and the results of analysis or the action performed. A channel report becomes pending when one of the following conditions has occurred:

1. Channel-subsystem recovery has been completed. The channel-subsystem recovery may have been initiated with no prior notice to the program or may have been a result of a condition previously reported to the program.

2. The function specified by RESET CHANNEL PATH has been completed.

A channel report may also become pending under other conditions.

The channel-report words which make up the channel report may be cleared, one at a time, by execution of the instruction STORE CHANNEL REPORT WORD, which is described in Chapter 14, "I/O Instructions."

Bit 9 is meaningless when channel-subsystem damage is reported.

Channel report pending is a floating repressible condition. It is masked by the channel-
report-pending subclass-mask bit, which is in bit position 35 of control register 14.

**Service-Processor Damage**

Bit 10 (SP), when one, indicates that damage has occurred to the service processor. Service-processor damage may be made pending at all CPUs in the configuration, or it may be detected independently by each CPU. The presence and extent of reporting service-processor damage depend on the model.

Service-processor damage is a repressible condition and has no subclass-mask bit.

**Channel-Subsystem Damage**

Bit 11 (CK), when one, indicates that an error or malfunction has occurred in the channel subsystem, or that the channel subsystem is in the check-stop state. The channel subsystem enters the check-stop state when a malfunction occurs which is so severe that the channel subsystem cannot continue, or if power is lost in the channel subsystem.

Channel-subsystem damage is a floating repressible condition and has no subclass-mask bit.

**Subclass Modifiers**

Bits 14 (B), 34 (DA), and 44 (AP) of the machine-check-interruption code act as modifiers to the subclass bits.

**Backed Up**

Bit 14 (B), when one, indicates that the point of interruption is at a checkpoint before the point of error. This bit is meaningful only when the instruction-processing-damage bit, bit 1, is also set to one. The presence and extent of the capability to indicate a backed-up condition depend on the model.

**Delayed Access Exception**

Bit 34 (DA), when one, indicates that an access exception was detected during a storage access using DAT when no such exception was detected by an earlier test for access exceptions.

Bit 34 is a modifier to instruction-processing damage (bit 1) and is meaningful only when bit 1 of the machine-check-interruption code is one.

When bit 1 is zero, bit 34 has no meaning. The presence and extent of reporting delayed access exception depend on the model.

**Programming Note:** The occurrence of a delayed access exception normally indicates that the program is using an improper procedure to update the DAT tables.

**Ancillary Report**

Bit 44 (AP), when one, indicates that a malfunction of a system component has occurred which has been recognized previously or which has affected the activities of multiple system elements such as CPUs and subchannels. When the malfunction affects the activities of multiple elements, an ancillary-report condition is recognized for all of the affected elements except one. This bit, when zero, indicates that this malfunction of a system component has not been recognized previously. This bit is meaningful for all conditions indicated by either the machine-check-interruption code or the external-damage code.

Depending on the model, recognition of an ancillary-report condition may not be provided, or it may not be provided for all system malfunctions. When ancillary-report recognition is not provided, bit 44 is set to zero.

**Synchronous Machine-Check-Interruption Conditions**

The instruction-processing damage and backed-up bits, bits 1 and 14 of the machine-check-interruption code, identify, in combination, two conditions.

<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 14</th>
<th>Name of Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Processing damage</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Processing backup</td>
</tr>
</tbody>
</table>

**Processing Backup**

The processing-backup condition indicates that the point of interruption is prior to the point, or points, of error. This is a nullifying exigent condition. When all of the other CPU-related-damage subclasses and modifiers of the machine-check-interruption code are zero, and certain validity bits associated with CPU status are indicated as valid, then the machine has successfully
returned to a checkpoint prior to the malfunction, and no damage has yet occurred to the CPU.

The subclass bits which must be zero for no damage to have occurred are as follows:

**MCIC**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>System damage</td>
</tr>
<tr>
<td>4</td>
<td>Timing-facility damage</td>
</tr>
</tbody>
</table>

The delayed-access-exception subclass-modifier bit, MCIC bit 34, must be zero for no damage to have occurred.

The validity bits in the machine-check-interruption code which must be one for no damage to have occurred are as follows:

**MCIC**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>PSW MWP bits</td>
</tr>
<tr>
<td>21</td>
<td>PSW mask and key</td>
</tr>
<tr>
<td>22</td>
<td>PSW program mask and condition code</td>
</tr>
<tr>
<td>23</td>
<td>PSW instruction address</td>
</tr>
<tr>
<td>27</td>
<td>Floating-point registers</td>
</tr>
<tr>
<td>28</td>
<td>General registers</td>
</tr>
<tr>
<td>29</td>
<td>Control registers</td>
</tr>
<tr>
<td>31</td>
<td>Storage logical validity (result fields within current checkpoint interval)</td>
</tr>
<tr>
<td>33</td>
<td>Access registers</td>
</tr>
<tr>
<td>42</td>
<td>TOD programmable register</td>
</tr>
<tr>
<td>43</td>
<td>Floating-point-control register</td>
</tr>
<tr>
<td>46</td>
<td>CPU timer</td>
</tr>
<tr>
<td>47</td>
<td>Clock comparator</td>
</tr>
</tbody>
</table>

**Programming Note:** The processing-backup condition is reported rather than system recovery to indicate that a malfunction or failure stands in the way of continued operation of the CPU. The malfunction has not been circumvented, and damage would have occurred if instruction processing had continued.

**Processing Damage**
The processing-damage condition indicates that damage has occurred to the instruction processing of the CPU. The point of interruption is a point beyond some or all of the points of damage. Processing damage is a terminating exigent condition; therefore, the contents of result fields may be unpredictable and still indicated as valid.

Processing damage may include malfunctions in program-event recording, monitor call, tracing, access-register translation, and dynamic address translation. Processing damage causes any supervisor-call-interruption condition and program-interruption condition to be discarded. However, the contents of the old PSW and interruption-code locations for these interruptions may be set to unpredictable values.

**Storage Errors**

Bits 16-18 of the machine-check-interruption code are used to indicate an invalid CBC or a near-valid CBC detected in main storage or an invalid CBC in a storage key. Bit 19, storage degradation, may be indicated concurrently with bit 17. The failing-storage-address field, when indicated as valid, identifies a location within the storage checking block containing the error, or, for storage-key error uncorrected, within the block associated with the storage key. Bit 32, indirect storage error, may be set to one to indicate that the location designated by the failing-storage address is not the original source of the error.

The storage-error-uncorrected and storage-key-error-uncorrected bits do not in themselves indicate the occurrence of damage because the error detected may not have affected a result. The portion of the configuration affected by an invalid CBC is indicated in the subclass field of the machine-check-interruption code.

Storage errors detected for a channel program, when indicated as I/O-error conditions, may also be reported as system recovery. CBC errors that occur in storage or in the storage key and that are detected on prefetched or unused data for a CPU program may or may not be reported, depending on the model.

**Storage Error Uncorrected**

Bit 16 (SE), when one, indicates that a checking block in main storage contained invalid CBC and that the information could not be corrected. The contents of the checking block in main storage have not been changed. The location reported may have been accessed or prefetched for this CPU or another CPU or a channel program, or it may have been accessed as the result of a model-dependent storage access.
Storage Error Corrected
Bit 17 (SC), when one, indicates that a checking block in main storage contained near-valid CBC and that the information has been corrected before being used. Depending on the model, the contents of the checking block in main storage may or may not have been restored to valid CBC. The location reported may have been accessed or prefetched for this CPU or for another CPU or for a channel program, or it may have been accessed as the result of a model-dependent storage access. The presence and extent of the storage-error-correction capability depend on the model. This indication may or may not be accompanied by an indication of storage degradation, bit 19 (DS).

Storage-Key Error Uncorrected
Bit 18 (KE), when one, indicates that a storage key contained invalid CBC and that the information could not be corrected. The contents of the checking block in the storage key have not been changed. The storage key may have been accessed or prefetched for this CPU or for another CPU or for a channel program, or it may have been accessed as the result of a model-dependent storage access.

Storage Degradation
Bit 19 (DS), when one, indicates that degradation of the recovery characteristics has occurred for the 4K-byte block reported by the failing-storage address.

Storage degradation indicates that although the associated storage error has been corrected, there are solid failures associated with the storage block (or with its associated key) that cause the correction process to take a substantial amount of time, and that if an additional error occurs in the block, the error may not be correctable or may go undetected. Thus, this bit indicates that use of the indicated block of storage should be avoided, if possible.

The indication of storage degradation has meaning only when failing-storage-address validity, MCIC bit 24, is also one. The presence and extent of reporting storage degradation depend on the model.

Programming Note: Because storage degradation is normally reported with system recovery, the recovery subclass mask, bit 36 of control register 14, should be set to one in order for storage degradation to be indicated.

Indirect Storage Error
Bit 32 (IE), when one, indicates that the physical main-storage location identified by the failing-storage address is not the original source of the error. Instead, the error originated in another level of the storage hierarchy and has been propagated to the current physical-storage portion of the storage hierarchy. Bit 32 is meaningful only when bit 16 or 18 (storage error uncorrected or storage-key error uncorrected) of the machine-check-interruption code is one. When bits 16 and 18 are both zeros, bit 32 has no meaning.

For errors originating outside the storage hierarchy, the attempt to store is rejected, and the appropriate error indication is presented. When an error is detected during implicit movement of information inside the storage hierarchy, the action is not rejected and reported in this manner because the movement may be asynchronous and may be initiated as the result of an attempt to access completely unrelated information. Instead, errors in the contents of the source during implicit moving of information from one portion of the storage hierarchy to another may be preserved in the target area by placing a special invalid CBC in the checking block associated with the target location. These propagated errors, when detected later, are reported as indirect storage errors. The original source of such an error may have been in a cache associated with an I/O processor or a CPU, or the error may have been the result of a data-path failure in transmitting data from one portion of the storage hierarchy to another. Additionally, a propagated error may be generated during the movement of data from one physical portion of storage to another as the result of a storage-reconfiguration action.

The presence and extent of reporting indirect storage error depend on the model.

Programming Note: See the programming notes under TEST BLOCK in Chapter 10, “Control Instructions” for the action which should be taken after storage errors are reported.
Machine-Check Interruption-Code Validity Bits

Bits 20-24, 26-29, 31, 33, 42, 43, 46, and 47 of the machine-check-interruption code are validity bits. Each bit indicates the validity of a particular field in storage. With the exception of the storage-logical-validity bit (bit 31), each bit is associated with a field stored during the machine-check interruption. When a validity bit is one, it indicates that the saved value placed in the corresponding storage field is valid with respect to the indicated point of interruption and that no error was detected when the data was stored.

When a validity bit is zero, one or more of the following conditions may have occurred: the original information was incorrect, the original information had invalid CBC, additional malfunctions were detected while storing the information, or none or only part of the information was stored. Even though the information is unpredictable, the machine attempts, when possible, to place valid CBC in the storage field and thus reduce the possibility of additional machine checks being caused.

The validity bits for the floating-point registers, general registers, control registers, access registers, TOD programmable register, floating-point control register, CPU timer, and clock comparator indicate the validity of the saved value placed in the corresponding save area. The information in these registers after the machine-check interruption is not necessarily correct even when the correct value has been placed in the save area and the validity bit set to one. The use of the registers and the operation of the facility associated with the control registers, floating-point control register, TOD programmable register, CPU timer, and clock comparator are unpredictable until these registers are validated. (See “Invalid CBC in Registers” on page 11-10)

PSW-MWP Validity
Bit 20 (WP), when one, indicates that bits 12-15 of the machine-check old PSW are correct.

PSW Mask and Key Validity
Bit 21 (MS), when one, indicates that the system mask, PSW key, and miscellaneous bits of the machine-check old PSW are correct. Specifically, this bit covers bits 0-11, 16, 17, 24-30, and 33-63 of the PSW.

PSW Program-Mask and Condition-Code Validity
Bit 22 (PM), when one, indicates that the program mask and condition code of the machine-check old PSW are correct.

PSW-Instruction-Address Validity
Bit 23 (IA), when one, indicates that the addressing-mode and instruction-address bits, bits 31, 32, and 64-127, of the machine-check old PSW are correct.

Failing-Storage-Address Validity
Bit 24 (FA), when one, indicates that a correct failing-storage address has been stored at real location 248-255 after a storage-error-uncorrected, storage-key-error-uncorrected, or storage-error-corrected condition has occurred. The presence and extent of the capability to identify the failing-storage location depend on the model. When no such errors are reported, that is, bits 16-18 of the machine-check-interruption code are zeros, the failing-storage address is meaningless, even though it may be indicated as valid.

External-Damage-Code Validity
Bit 26 (EC), when one, and provided that bit 5, external damage, is also one, indicates that a valid external-damage code has been stored in the word at real location 244. When bit 5 is zero, bit 26 has no meaning.

Floating-Point-Register Validity
Bit 27 (FP), when one, indicates that the contents of the floating-point-register save area at real locations 4608-4735 reflect the correct state of the floating-point registers at the point of interruption.

General-Register Validity
Bit 28 (GR), when one, indicates that the contents of the general-register save area at real locations 4736-4863 reflect the correct state of the general registers at the point of interruption.

Control-Register Validity
Bit 29 (CR), when one, indicates that the contents of the control-register save area at real locations 4992-5119 reflect the correct state of the control registers at the point of interruption.
Storage Logical Validity
Bit 31 (ST), when one, indicates that the storage locations, the contents of which are modified by the instructions being executed, contain the correct information relative to the point of interruption. That is, all stores before the point of interruption are completed, and all stores, if any, after the point of interruption are suppressed. When a store before the point of interruption is suppressed because of an invalid CBC, the storage-logical-validity bit may be indicated as one, provided that the invalid CBC has been preserved as invalid.

When instruction-processing damage is indicated but processing backup is not indicated, the storage-logical-validity bit has no meaning.

Storage logical validity reflects only the instruction-processing activity and does not reflect errors in the state of storage as the result of either I/O operations or the storing of the old PSW and other interruption information.

Access-Register Validity
Bit 33 (AR), when one, indicates that the contents of the access-register save area at real locations 4928-4991 reflect the correct state of the access registers at the point of interruption.

TOD-Programmable-Register Validity
Bit 42 (PR), when one, indicates that the contents of the TOD-programmable-register save area at real locations 4900-4903 reflect the correct state of the TOD programmable register at the point of interruption.

Floating-Point-Control-Register Validity
Bit 43 (FC), when one, indicates that the contents of the floating-point-control-register save area at real locations 4892-4895 reflect the correct state of the floating-point-control register at the point of interruption.

CPU-Timer Validity
Bit 46 (CT), when one, indicates that the CPU timer is not in error and that the contents of the CPU-timer save area at real locations 4904-4911 reflect the correct state of the CPU timer at the time the interruption occurred.

Clock-Comparator Validity
Bit 47 (CC), when one, indicates that the clock comparator is not in error, that the contents of the clock-comparator save area at real locations 4913-4919 reflect the correct state of the clock comparator at the time the interruption occurred, and that zeros have been stored at real location 4912.

Programming Note: The validity bits must be used in conjunction with the subclass bits and the backed-up bit in order to determine the extent of the damage caused by a machine-check condition. No damage has occurred to the system when all of the following are true:
- The four PSW-validity bits, the six register-validity bits, the two timing-facility-validity bits, and the storage-logical-validity bit are all ones.
- Subclass bits 0, 4, 5, 10, and 11 are zeros.
- The instruction-processing-damage bit is zero or, if one, the backed-up bit is also one.
- The delayed-access-exception bit is zero.

Machine-Check Extended Interruption Information
As part of the machine-check interruption, in some cases, extended interruption information is placed in fixed areas assigned in storage. The contents of registers associated with the CPU are placed in register-save areas. For external damage, additional information is provided for some models by storing an external-damage code. When storage error uncorrected, storage error corrected, or storage-key error uncorrected is indicated, the failing-storage address is saved.

Each of these fields has associated with it a validity bit in the machine-check-interruption code. If, for any reason, the machine cannot store the proper information in the field, the associated validity bit is set to zero.

Register-Save Areas
As part of the machine-check interruption, the current contents of the CPU registers, except for the prefix register and the TOD clock, are stored in eight register-save areas assigned in storage. Each of these areas has associated with it a validity bit in the machine-check-interruption code.
If, for any reason, the machine cannot store the proper information in the field, the associated validity bit is set to zero.

The following are the eight sets of registers and the real locations in storage where their contents are saved during a machine-check interruption.

<table>
<thead>
<tr>
<th>Locations</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>4608-4735</td>
<td>Floating-point registers 0-15</td>
</tr>
<tr>
<td>4736-4863</td>
<td>General registers 0-15</td>
</tr>
<tr>
<td>4892-4895</td>
<td>Floating-point-control register</td>
</tr>
<tr>
<td>4900-4903</td>
<td>TOD programmable register</td>
</tr>
<tr>
<td>4904-4911</td>
<td>CPU timer</td>
</tr>
<tr>
<td>4913-4919</td>
<td>Clock comparator</td>
</tr>
<tr>
<td>4928-4991</td>
<td>Access registers 0-15</td>
</tr>
<tr>
<td>4992-5119</td>
<td>Control registers 0-15</td>
</tr>
</tbody>
</table>

**External-Damage Code**

The word at real location 244 is the external-damage code. This field, when implemented and indicated as valid, describes the cause of external damage. The field is valid only when the external-damage bit and the external-damage-code-validity bit (bits 5 and 26 in the machine-check-interruption code) are both ones. The presence and extent of reporting an external-damage code depend on the model.

The external-damage code has the following format:

```
/ /  / S  0  0  0  0 
0 19 31
```

**ETR Sync Check (SC):** Bit 19, when one, indicates that bits 32 through the rightmost incremented bit of a running clock are not in synchronism with the same bits of the ETR.

If the condition happens more than once before the interruption occurs, the condition is generated only once. The condition is generated for all CPUs in the configuration, and the condition for a CPU is cleared by the interruption taken by the CPU.

**Reserved:** Bits 0-7, 10-16, and 21-31 are reserved for future expansion and are always set to zero.

**Failing-Storage Address**

When storage error uncorrected, storage error corrected, or storage-key error uncorrected is indicated in the machine-check-interruption code, the associated address, called the failing-storage address, is stored at real locations 248-255. The field is valid only if the failing-storage-address validity bit, bit 24 of the machine-check-interruption code, is one.

In the case of storage errors, the failing-storage address may designate any byte within the checking block. For storage-key error uncorrected, the failing-storage address may designate any address within the block of storage associated with the storage key that is in error. When an error is detected in more than one location before the interruption, the failing-storage address may designate any of the failing locations. The address stored is an absolute address; that is, the value stored is the address that is used to reference storage after dynamic address translation and prefixing have been applied.

**Handling of Machine-Check Conditions**

**Floating Interruption Conditions**

An interruption condition which is made available to any CPU in a multiprocessing configuration is called a floating interruption condition. The first CPU that accepts the interruption clears the interruption condition, and it is no longer available to any other CPU in the configuration.

Floating interruption conditions include service-signal external-interruption and I/O-interruption conditions. Two machine-check-interruption conditions, channel report pending and channel-subsystem damage, are floating interruption conditions. Depending on the model, some machine-check-interruption conditions associated with system recovery and warning may also be floating interruption conditions.

A floating interruption is presented to the first CPU in the configuration which is enabled for the interruption condition and can accept the interruption. A CPU cannot accept the interruption when the CPU is in the check-stop state, has an invalid
prefix, is performing an unending string of interruptions due to a PSW-format error of the type that is recognized early, or is in the stopped state. However, a CPU with the rate control set to instruction step can accept the interruption when the start key is activated.

Programming Note: When a CPU enters the check-stop state in a multiprocessing configuration, the program on another CPU can determine whether a floating interruption may have been reported to the failing CPU and then lost. This can be accomplished if the interruption program places zeros in the real storage locations containing old PSWs and interruption codes after the interruption has been handled (or has been moved into another area for later processing). After a CPU enters the check-stop state, the program on another CPU can inspect the old-PSW and interruption-code locations of the failing CPU. A nonzero value in an old PSW or interruption code indicates that the CPU has been interrupted but the program did not complete the handling of the interruption.

Floating Machine-Check- Interruption Conditions
Floating machine-check-interruption conditions are reset only by the manually initiated resets through the operator facilities. When a machine check occurs which prohibits completion of a floating machine-check interruption, the interruption condition is no longer considered a floating interruption condition, and system damage is indicated.

Floating I/O Interruptions
The detection of a machine malfunction by the channel subsystem, while in the process of presenting an I/O-interruption request for a floating I/O interruption, may be reported as channel report pending or as channel-subsystem damage. Detection of a machine malfunction by a CPU, while in the process of accepting a floating I/O interruption, is reported as system damage.

Machine-Check Masking
All machine-check interruptions are under control of the machine-check mask, PSW bit 13. In addition, some machine-check conditions are controlled by subclass masks in control register 14.

The exigent machine-check conditions (system damage and instruction-processing damage) are controlled only by the machine-check mask, PSW bit 13. When PSW bit 13 is one, an exigent condition causes a machine-check interruption. When PSW bit 13 is zero, the occurrence of an exigent machine-check condition causes the CPU to enter the check-stop state.

The repressible machine-check conditions, except channel-subsystem damage and service-processor damage, are controlled both by the machine-check mask, PSW bit 13, and by five subclass-mask bits in control register 14. If PSW bit 13 is one and one of the subclass-mask bits is one, the associated condition initiates a machine-check interruption. If a subclass-mask bit is zero, the associated condition does not initiate an interruption but is held pending. However, when a machine-check interruption is initiated because of a condition for which the CPU is enabled, those conditions for which the CPU is not enabled may be presented along with the condition which initiates the interruption. All conditions presented are then cleared.

Control register 14 contains mask bits that specify whether certain conditions can cause machine-check interruptions. It has the following format:

```
```

Bits 35-39 of control register 14 are the subclass masks for repressible machine-check conditions. In addition, bit 32 of control register 14 is initialized to one but is otherwise ignored by the machine.

Programming Note: The program should avoid, whenever possible, operating with PSW bit 13, the machine-check mask, set to zero, since any exigent machine-check condition which is recognized during this situation will cause the CPU to enter the check-stop state. In particular, the program should avoid executing I/O instructions or allowing I/O interruptions with PSW bit 13 zero.

Channel-Report-Pending Subclass Mask
Bit 35 (CM) of control register 14 controls channel-report-pending interruption conditions. This bit is initialized to zero.
Recovery Subclass Mask
Bit 36 (RM) of control register 14 controls system-recovery interruption conditions. This bit is initialized to zero.

Degradation Subclass Mask
Bit 37 (DM) of control register 14 controls degradation interruption conditions. This bit is initialized to zero.

External-Damage Subclass Mask
Bit 38 (EM) of control register 14 controls timing-facility-damage and external-damage interruption conditions. This bit is initialized to one.

Warning Subclass Mask
Bit 39 (WM) of control register 14 controls warning interruption conditions. This bit is initialized to zero.

Machine-Check Logout
As part of the machine-check interruption, some models may place model-dependent information in the fixed-logout area. This area is 16 bytes in length and starts at real location 4864.

Summary of Machine-Check Masking
A summary of machine-check masking is given in Figure 11-4 and Figure 11-5 on page 11-26.

<table>
<thead>
<tr>
<th>Machine-Check Condition</th>
<th>Sub-Condition Mask</th>
<th>Action when CPU Disabled for Subclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 System damage</td>
<td>-</td>
<td>Check stop</td>
</tr>
<tr>
<td>1 Instruction-processing damage</td>
<td>RM</td>
<td>Check stop</td>
</tr>
<tr>
<td>2 System recovery</td>
<td>EM</td>
<td>P</td>
</tr>
<tr>
<td>4 Timing-facility damage</td>
<td>EM</td>
<td>P</td>
</tr>
<tr>
<td>5 External damage</td>
<td>DM</td>
<td>Y</td>
</tr>
<tr>
<td>7 Degradation</td>
<td>WM</td>
<td>P</td>
</tr>
<tr>
<td>8 Warning</td>
<td>WM</td>
<td>P</td>
</tr>
<tr>
<td>9 Channel report pending</td>
<td>CM</td>
<td>P</td>
</tr>
<tr>
<td>10 Service-processor damage</td>
<td>-</td>
<td>P</td>
</tr>
<tr>
<td>11 Channel-subsystem damage</td>
<td>-</td>
<td>P</td>
</tr>
</tbody>
</table>

Explanation:
- The condition does not have a subclass mask.
- P Indication is held pending.
- Y Indication may be held pending or may be discarded.
- CM Channel-report-pending subclass mask (bit 35 of CR14).
- DM Degradation subclass mask (bit 37 of CR14).
- EM External-damage subclass mask (bit 38 of CR14).
- RM Recovery subclass mask (bit 36 of CR14).
- WM Warning subclass mask (bit 39 of CR14).

Figure 11-4. Machine-Check-Condition Masking
<table>
<thead>
<tr>
<th>Bit Description</th>
<th>Control Register 14 Bit Position</th>
<th>State of Bit on Initial CPU Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel-report-pending subclass mask</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>Recovery subclass mask</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>Degradation subclass mask</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>External-damage subclass mask</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>Warning subclass mask</td>
<td>39</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 11-5. Machine-Check Control-Register Bits
## Chapter 12. Operator Facilities

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Operation</td>
<td>12-1</td>
</tr>
<tr>
<td>Basic Operator Facilities</td>
<td>12-1</td>
</tr>
<tr>
<td>Address-Compare Controls</td>
<td>12-1</td>
</tr>
<tr>
<td>Alter-and-Display Controls</td>
<td>12-2</td>
</tr>
<tr>
<td>Architectural-Mode Indicator</td>
<td>12-2</td>
</tr>
<tr>
<td>Architectural-Mode-Selection Controls</td>
<td>12-2</td>
</tr>
<tr>
<td>Check-Stop Indicator</td>
<td>12-3</td>
</tr>
<tr>
<td>IML Controls</td>
<td>12-3</td>
</tr>
<tr>
<td>Interrupt Key</td>
<td>12-3</td>
</tr>
<tr>
<td>Load Indicator</td>
<td>12-3</td>
</tr>
<tr>
<td>Load-Clear Key</td>
<td>12-3</td>
</tr>
<tr>
<td>Load-Clear-List-Directed Key</td>
<td>12-3</td>
</tr>
<tr>
<td>Load-Normal Key</td>
<td>12-3</td>
</tr>
<tr>
<td>Load-with-Dump Key</td>
<td>12-3</td>
</tr>
<tr>
<td>Load-Unit-Address Controls</td>
<td>12-4</td>
</tr>
<tr>
<td>Manual Indicator</td>
<td>12-4</td>
</tr>
<tr>
<td>Power Controls</td>
<td>12-4</td>
</tr>
<tr>
<td>Rate Control</td>
<td>12-4</td>
</tr>
<tr>
<td>Restart Key</td>
<td>12-4</td>
</tr>
<tr>
<td>Start Key</td>
<td>12-5</td>
</tr>
<tr>
<td>Stop Key</td>
<td>12-5</td>
</tr>
<tr>
<td>Store-Status Key</td>
<td>12-5</td>
</tr>
<tr>
<td>System-Reset-Clear Key</td>
<td>12-5</td>
</tr>
<tr>
<td>System-Reset-Normal Key</td>
<td>12-5</td>
</tr>
<tr>
<td>Test Indicator</td>
<td>12-5</td>
</tr>
<tr>
<td>TOD-Clock Control</td>
<td>12-6</td>
</tr>
<tr>
<td>Wait Indicator</td>
<td>12-6</td>
</tr>
<tr>
<td>Multiprocessing Configurations</td>
<td>12-6</td>
</tr>
</tbody>
</table>

### Manual Operation

The operator facilities provide functions for the manual operation and control of the machine. The functions include operator-to-machine communication, indication of machine status, control over the setting of the TOD clock, initial program loading, resets, and other manual controls for operator intervention in normal machine operation.

A model may provide additional operator facilities which are not described in this chapter. Examples are the means to indicate specific error conditions in the equipment, to change equipment configurations, and to facilitate maintenance. Furthermore, controls covered in this chapter may have additional settings which are not described here. Such additional facilities and settings may be described in the appropriate System Library publication.

Most models provide, in association with the operator facilities, a console device which may be used as an I/O device for operator communication with the program; this console device may also be used to implement some or all of the facilities described in this chapter.

The operator facilities may be implemented on different models in various technologies and configurations. On some models, more than one set of physical representations of some keys, controls, and indicators may be provided, such as on multiple local or remote operating stations, which may be effective concurrently.

A machine malfunction that prevents a manual operation from being performed correctly, as defined for that operation, may cause the CPU to enter the check-stop state or give some other indication to the operator that the operation has failed. Alternatively, a machine malfunction may cause a machine-check-interruption condition to be recognized.

### Basic Operator Facilities

#### Address-Compare Controls

The address-compare controls provide a way to stop the CPU when a preset address matches the address used in a specified type of main-storage reference.

One of the address-compare controls is used to set up the address to be compared with the storage address.

Another control provides at least two positions to specify the action, if any, to be taken when the address match occurs:

1. The normal position disables the address-compare operation.
2. The stop position causes the CPU to enter the stopped state on an address match. When the control is in this setting, the test indicator is on. Depending on the model and the type
of reference, pending I/O, external, and machine-check interruptions may or may not be taken before entering the stopped state.

A third control may specify the type of storage reference for which the address comparison is to be made. A model may provide one or more of the following positions, as well as others:

1. The any position causes the address comparison to be performed on all storage references.
2. The data-store position causes address comparison to be performed when storage is addressed to store data.
3. The I/O position causes address comparison to be performed when storage is addressed by the channel subsystem to transfer data or to fetch a channel-command or indirect-data-address word. Whether references to the measurement block, interruption-response block, channel-path-status word, channel-report word, subchannel-status word, subchannel-information block, and operation-request block cause a match to be indicated depends on the model.
4. The instruction-address position causes address comparison to be performed when storage is addressed to fetch an instruction. The rightmost bit of the address setting may or may not be ignored. The match is indicated only when the first byte of the instruction is fetched from the selected location. It depends on the model whether a match is indicated when fetching the target instruction of EXECUTE.

Depending on the model and the type of reference, address comparison may be performed on virtual, real, or absolute addresses, and it may be possible to specify the type of address.

In a multiprocessing configuration, it depends on the model whether the address setting applies to one or all CPUs in the configuration and whether an address match causes one or all CPUs in the configuration to stop.

### Alter-and-Display Controls

The operator facilities provide controls and procedures to permit the operator to alter and display the contents of locations in storage, the storage keys, the general, floating-point, floating-point-control, access, and control registers, the prefix, and the PSW.

Before alter-and-display operations may be performed, the CPU must first be placed in the stopped state. During alter-and-display operations, the manual indicator may be turned off temporarily, and the start and restart keys may be inoperative.

Addresses used to select storage locations for alter-and-display operations are real addresses. The capability of specifying logical, virtual, or absolute addresses may also be provided.

### Architectural-Mode Indicator

The architectural-mode indicator shows the architectural mode of operation (the ESA/390 mode, z/Architecture mode, or some other mode) selected by the last architectural-mode-selection operation and by the last SIGNAL PROCESSOR set-architecture order or the last reset that determined the mode.

### Architectural-Mode-Selection Controls

The architectural-mode-selection controls provide for the selection of either the ESA/390 architectural mode of operation or, possibly, some other architectural mode of operation. (The z/Architecture mode is selected from the ESA/390 mode by the SIGNAL PROCESSOR set-architecture order.) Depending on the model, the architectural-mode selection may be provided as part of the IML operation or may be a separate operation.

As part of the architectural-mode-selection process, all CPUs and the associated channel-subsystem components in a particular configuration are placed in the same architectural mode.
Check-Stop Indicator

The check-stop indicator is on when the CPU is in the check-stop state. Reset operations normally cause the CPU to leave the check-stop state and thus turn off the indicator. The manual indicator may also be on in the check-stop state.

IML Controls

The IML controls provided with some models perform initial machine loading (IML), which is the loading of licensed internal code into the machine. The IML operation, when provided, may be used to select the ESA/390 mode or, possibly, some other mode of operation.

When the IML operation is completed, the state of the affected CPUs, channel subsystem, main storage, and operator facilities is the same as if a power-on reset had been performed, except that the value and state of the TOD clock are not changed. The contents of expanded storage may have been cleared to zeros with valid checking-block code or may have remained unchanged, depending on the model.

The IML controls are effective while the power is on.

Interrupt Key

When the interrupt key is activated, an external-interruption condition indicating the interrupt key is generated. (See "Interrupt Key" on page 6-12.)

The interrupt key is effective when the CPU is in the operating or stopped state. It depends on the model whether the interrupt key is effective when the CPU is in the load state.

Load Indicator

The load indicator is on during CCW-type initial program loading, indicating that the CPU is in the load state. The indicator goes on for a particular CPU when the load-clear or load-normal key is activated for that CPU and the corresponding operation is started. It goes off after the new PSW is loaded successfully. For details, see "Initial Program Loading" on page 4-54.

Load-Clear Key

Activating the load-clear key causes a reset operation to be performed and CCW-type initial program loading to be started by using the I/O device designated by the load-unit-address controls. Clear reset is performed on the configuration. For details, see "Resets" on page 4-47 and "Initial Program Loading" on page 4-54.

The load-clear key is effective when the CPU is in the operating, stopped, load, or check-stop state.

Load-Clear-List-Directed Key

Activating the load-clear-list-directed key causes a reset operation to be performed and list-directed initial program loading to be started using the I/O device designated by the list-directed IPL parameters. Clear reset is performed on the configuration. For details, see "Resets" on page 4-47 and "List-Directed IPL" on page 17-20.

The load-clear-list-directed key is effective when the CPU is in the operating, stopped, load, or check-stop state.

Load-Normal Key

Activating the load-normal key causes a reset operation to be performed and CCW-type initial program loading to be started by using the I/O device designated by the load-unit-address controls. Initial CPU reset is performed on the CPU for which the load-normal key was activated, CPU reset is propagated to all other CPUs in the configuration, and a subsystem reset is performed on the remainder of the configuration. For details, see "Resets" on page 4-47 and "Initial Program Loading" on page 4-54.

The load-normal key is effective when the CPU is in the operating, stopped, load, or check-stop state.

Load-with-Dump Key

Activating the load-with-dump key causes a reset operation to be performed and list-directed initial program loading to be started using the I/O devices designated by the list-directed IPL parameters. A store-status operation followed by an initial CPU reset is performed on the CPU for
which the load-with-dump key was activated, CPU reset is propagated to all other CPUs in the configuration, and a subsystem reset is performed on the remainder of the configuration. For details, see "Resets" on page 4-47 and "List-Directed IPL" on page 17-20.

The load-with-dump key is effective when the CPU is in the operating, stopped, load, or check-stop state.

Load-Unit-Address Controls

The load-unit-address controls specify four hexadecimal digits that provide the device number used for CCW-type initial program loading. For details, see "Initial Program Loading" on page 4-54.

Manual Indicator

The manual indicator is on when the CPU is in the stopped state. Some functions and several manual controls are effective only when the CPU is in the stopped state.

Power Controls

The power controls are used to turn the power on and off.

The CPUs, storage, channel subsystem, operator facilities, and I/O devices may all have their power turned on and off by common controls, or they may have separate power controls. When a particular unit has its power turned on, that unit is reset. The sequence is performed so that no instructions or I/O operations are performed until explicitly specified. The controls may also permit power to be turned on in stages, but the machine does not become operational until power on is complete.

When the power is completely turned on, an IML operation is performed on models which have an IML function. A power-on reset is then initiated (see "Resets" on page 4-47). It depends on the model whether the architectural mode of operation can be selected when the power is turned on, or whether the mode-selection controls have to be used to change the mode after the power is on.

Rate Control

The setting of the rate control determines the effect of the start function and the manner in which instructions are executed.

The rate control has at least two positions. The normal position is the process position. Another position is the instruction-step position. When the rate control is set to the process position and the start function is performed, the CPU starts operating at normal speed. When the rate control is set to the instruction-step position and the wait-state bit is zero, one instruction or, for interruptible instructions, one unit of operation is executed, and all pending allowed interruptions are taken before the CPU returns to the stopped state. When the rate control is set to the instruction-step position and the wait-state bit is one, no instruction is executed, but all pending allowed interruptions are taken before the CPU returns to the stopped state. For details, see "Stopped, Operating, Load, and Check-Stop States" on page 4-1.

The test indicator is on while the rate control is not set to the process position.

If the setting of the rate control is changed while the CPU is in the operating or load state, the results are unpredictable.

Restart Key

Activating the restart key initiates a restart interruption. (See "Restart Interruption" on page 6-47)

The restart key is effective when the CPU is in the operating or stopped state. The key is not effective when the CPU is in the check-stop state. It depends on the model whether the restart key is effective when any CPU in the configuration is in the load state.

The effect is unpredictable when the restart key is activated while any CPU in the configuration is in the load state. In particular, if the CPU performs a restart interruption and enters the operating state while another CPU is in the load state, operations such as I/O instructions, the SIGNAL PROCESSOR instruction, and the INVALIDATE PAGE TABLE ENTRY instruction may not operate according to the definitions given in this publication.
Start Key

Activating the start key causes the CPU to perform the start function. (See “Stopped, Operating, Load, and Check-Stop States” on page 4-1.)

The start key is effective only when the CPU is in the stopped state. The effect is unpredictable when the stopped state has been entered by a reset.

Stop Key

Activating the stop key causes the CPU to perform the stop function. (See “Stopped, Operating, Load, and Check-Stop States” on page 4-1.)

The stop key is effective only when the CPU is in the operating state.

Operation Note: Activating the stop key has no effect when:

- An unending string of certain program or external interruptions occurs.
- The prefix register contains an invalid address.
- The CPU is in the load or check-stop state.

Store-Status Key

Activating the store-status key initiates a store-status operation. (See “Store Status” on page 4-55)

The store-status key is effective only when the CPU is in the stopped state.

Operation Note: The store-status operation may be used in conjunction with a standalone dump program for the analysis of major program malfunctions. For such an operation, the following sequence would be called for:

1. Activation of the stop or system-reset-normal key
2. Activation of the store-status key
3. Activation of the load-normal key to enter a standalone dump program

The system-reset-normal key must be activated in step 1 when (1) the stop key is not effective because a continuous string of interruptions is occurring, (2) the prefix register contains an invalid address, or (3) the CPU is in the check-stop state.

System-Reset-Clear Key

Activating the system-reset-clear key causes a clear-reset operation to be performed on the configuration. For details, see “Resets” on page 4-47.

The system-reset-clear key is effective when the CPU is in the operating, stopped, load, or check-stop state.

System-Reset-Normal Key

Activating the system-reset-normal key causes a CPU-reset operation and a subsystem-reset operation to be performed. In a multiprocessing configuration, a CPU reset is propagated to all CPUs in the configuration. For details, see the section “Resets” in Chapter 4, “Control.”

The system-reset-normal key is effective when the CPU is in the operating, stopped, load, or check-stop state.

Test Indicator

The test indicator is on when a manual control for operation or maintenance is in an abnormal position that can affect the normal operation of a program.

Setting the address-compare controls to the stop position or setting the rate control to the instruction-step position turns on the test indicator.

The test indicator may be on when one or more diagnostic functions under the control of DIAGNOSE are activated, or when other abnormal conditions occur.

The abnormal setting of a manual control causes the test indicator of the affected CPU to be turned on; however, in a multiprocessing configuration, the operation of other CPUs may be affected even though their test indicators are not turned on.

Operation Note: If a manual control is left in a setting intended for maintenance purposes, such an abnormal setting may, among other things, result in false machine-check indications or cause actual machine malfunctions to be ignored. It may
also alter other aspects of machine operation, including instruction execution, channel-subsystem operation, and the functioning of operator controls and indicators, to the extent that operation of the machine does not comply with that described in this publication.

**TOD-Clock Control**

When the TOD-clock control is not activated, that is, the control is set to the secure position, the state and value of the TOD clock are protected against unauthorized or inadvertent change by not permitting the instructions SET CLOCK or DIAGNOSE to change the state or value.

When the TOD-clock control is activated, that is, the control is set to the enable-set position, alteration of the clock state or value by means of SET CLOCK or DIAGNOSE is permitted. This setting is momentary, and the control automatically returns to the secure position.

If there is more than one physical representation of the TOD-clock control, the TOD clock is secure only if all TOD-clock controls in the configuration are set to the secure position.

**Wait Indicator**

The wait indicator is on when the wait-state bit in the current PSW is one. Instead of a wait indicator, a model may have a means of indicating a time-averaged value of the wait-state bit.

---

**Multiprocessing Configurations**

In a multiprocessing configuration, one of each of the following keys and controls is provided for each CPU: alter and display, interrupt, rate, restart, start, stop, and store status. The load-clear key, load-normal key, and load-unit-address controls are provided for each CPU capable of performing I/O operations. Alternatively, a single set of initial-program-loading keys and controls may be used together with a control to select the desired CPU.

There need not be more than one of each of the following keys and controls in a multiprocessing configuration: address compare, IML, power, system reset clear, system reset normal, and TOD clock.

One check-stop, manual, test, and wait indicator is provided for each CPU. A load indicator is provided only on a CPU capable of performing I/O operations. Alternatively, a single set of indicators may be switched to more than one CPU.

There need not be more than one architectural-mode indicator in a multiprocessing configuration.

In a system capable of reconfiguration, there must be a separate set of keys, controls, and indicators in each configuration.
Chapter 13. I/O Overview

The terms “input” and “output” are used to describe the transfer of data between I/O devices and main storage. An operation involving this kind of transfer is referred to as an I/O operation. The facilities used to control I/O operations are collectively called the channel subsystem. (I/O devices and their control units attach to the channel subsystem.) This chapter provides a brief description of the basic components and operation of the channel subsystem.

### Input/Output (I/O)

The terms “input” and “output” are used to describe the transfer of data between I/O devices and main storage. An operation involving this kind of transfer is referred to as an I/O operation. The facilities used to control I/O operations are collectively called the channel subsystem. (I/O devices and their control units attach to the channel subsystem.) This chapter provides a brief description of the basic components and operation of the channel subsystem.

### The Channel Subsystem

The channel subsystem directs the flow of information between I/O devices and main storage. It relieves CPUs of the task of communicating directly with I/O devices and permits data processing to proceed concurrently with I/O processing. The channel subsystem uses one or more channel paths as the communication link in managing the flow of information to or from I/O devices. As part of I/O processing, the channel subsystem also performs a path-management operation by testing for channel-path availability, chooses an available channel path, and initiates the performance of the I/O operation by the device.

Within the channel subsystem are subchannels. One subchannel is provided for and dedicated to each I/O device accessible to the program through the channel subsystem. Each subchannel provides information concerning the associated I/O device and its attachment to the channel subsystem. The subchannel also provides information concerning I/O operations and other functions involving the associated I/O device. The subchannel is the means by which the channel subsystem provides information about associated I/O devices to CPUs, which obtain this information by executing I/O instructions. The actual number of subchannels provided depends on the model and the configuration; the maximum addressability is 0-65,535.

I/O devices are attached through control units to the channel subsystem by means of channel paths. Control units may be attached to the channel subsystem by more than one channel path, and an I/O device may be attached to more than one control unit. In all, an individual I/O device may be accessible to the channel subsystem by as many as eight different channel paths via a subchannel, depending on the model and the configuration. The total number of channel paths provided by a channel subsystem depends on the model and the configuration; the maximum addressability is 0-255.

The performance of a channel subsystem depends on its use and on the system model in which it is implemented. Channel paths are provided with different data-transfer capabilities, and an I/O device designed to transfer data only at a specific rate (a magnetic-tape unit or a disk storage, for example) can operate only on a channel path that can accommodate at least this data rate.

The channel subsystem contains common facilities for the control of I/O operations. When these facilities are provided in the form of separate, autonomous equipment designed specifically to control I/O devices, I/O operations are completely overlapped with the activity in CPUs. The only main-storage cycles required by the channel sub-

<table>
<thead>
<tr>
<th>Input/Output (I/O)</th>
<th>Subchannel Number</th>
<th>Device Number</th>
<th>Device Identifier</th>
<th>Execution of I/O Operations</th>
<th>Start-Function Initiation</th>
<th>Path Management</th>
<th>Channel-Program Execution</th>
<th>Conclusion of I/O Operations</th>
<th>I/O Interruptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Channel Subsystem</td>
<td>13-1</td>
<td>13-5</td>
<td>13-5</td>
<td>13-6</td>
<td>13-6</td>
<td>13-6</td>
<td>13-7</td>
<td>13-8</td>
<td>13-9</td>
</tr>
<tr>
<td>Control Units</td>
<td>13-4</td>
<td>13-5</td>
<td>13-5</td>
<td>13-6</td>
<td>13-6</td>
<td>13-6</td>
<td>13-7</td>
<td>13-8</td>
<td>13-9</td>
</tr>
<tr>
<td>Channel-Path Identifier</td>
<td>13-5</td>
<td>13-5</td>
<td>13-5</td>
<td>13-6</td>
<td>13-6</td>
<td>13-6</td>
<td>13-7</td>
<td>13-8</td>
<td>13-9</td>
</tr>
</tbody>
</table>
system during I/O operations are those needed to transfer data and control information to or from the final locations in main storage, along with those cycles that may be required for the channel subsystem to access the subchannels when they are implemented as part of nonaddressable main storage. These cycles do not delay CPU programs, except when both the CPU and the channel subsystem concurrently attempt to reference the same main-storage area.

**Subchannels**

A subchannel provides the logical appearance of a device to the program and contains the information required for sustaining a single I/O operation. The subchannel consists of internal storage that contains information in the form of a CCW address, channel-path identifier, device number, count, status indications, and I/O-interruption-subclass code, as well as information on path availability and functions pending or being performed. I/O operations are initiated with a device by the execution of I/O instructions that designate the subchannel associated with the device.

Each device is accessible by means of one subchannel in each channel subsystem to which it is assigned during configuration at installation time. The device may be a physically identifiable unit or may be housed internal to a control unit. For example, in certain disk-storage devices, each actuator used in retrieving data is considered to be a device. In all cases, a device, from the point of view of the channel subsystem, is an entity that is uniquely associated with one subchannel and that responds to selection by the channel subsystem by using the communication protocols defined for the type of channel path by which it is accessible.

On some models, subchannels are provided in blocks. On these models, more subchannels may be provided than there are attached devices. Subchannels that are provided but do not have devices assigned to them are not used by the channel subsystem to perform any function and are indicated by storing the associated device-number-valid bit as zero in the subchannel-information block of the subchannel.

The number of subchannels provided by the channel subsystem is independent of the number of channel paths to the associated devices. For example, a device accessible through alternate channel paths still is represented by a single subchannel. Each subchannel is addressed by using a 16-bit binary subchannel number.

After I/O processing at the subchannel has been requested by the execution of START SUBCHANNEL, the CPU is released for other work, and the channel subsystem assembles or disassembles data and synchronizes the transfer of data bytes between the I/O device and main storage. To accomplish this, the channel subsystem maintains and updates an address and a count that describe the destination or source of data in main storage. Similarly, when an I/O device provides signals that should be brought to the attention of the program, the channel subsystem transforms the signals into status information and stores the information in the subchannel, where it can be retrieved by the program.

**Attachment of Input/Output Devices**

**Channel Paths**

The channel subsystem communicates with I/O devices by means of channel paths between the channel subsystem and control units. A control unit may be accessible by the channel subsystem by more than one channel path. Similarly, an I/O device may be accessible by the channel subsystem through more than one control unit, each having one or more channel paths to the channel subsystem.

Devices that are attached to the channel subsystem by multiple channel paths configured to a subchannel, may be accessed by the channel subsystem using any of the available channel paths. Similarly, a device having the dynamic-reconnection feature and operating in the multi-path mode can be initialized to operate such that the device may choose any of the available channel paths configured to the subchannel, when logically reconnecting to the channel subsystem to continue a chain of I/O operations.

The channel subsystem may contain more than one type of channel path. Examples of channel-path types used by the channel subsystem are the...
ESCON I/O interface, FICON I/O interface, FICON-converted I/O interface, and IBM System/360 and System/370 I/O interface. The term “serial-I/O interface” is used to refer to the ESCON I/O interface, the FICON I/O interface, and the FICON-converted I/O interface. The term “parallel-I/O interface” is used to refer to the IBM System/360 and System/370 I/O interface.


Depending on the type of channel path, the facilities provided by the channel path, and the I/O device, an I/O operation may occur in one of three modes, frame-multiplex mode, burst mode, or byte-multiplex mode.

In the frame-multiplex mode, the I/O device may stay logically connected to the channel path for the duration of the execution of a channel program. The facilities of a channel path capable of operating in the frame-multiplex mode may be shared by a number of concurrently operating I/O devices. In this mode the information required to complete an I/O operation is divided into frames that may be interleaved with frames from I/O operations for other I/O devices. During this period, multiple I/O devices are considered to be logically connected to the channel path.

In the burst mode, the I/O device monopolizes a channel path and stays logically connected to the channel path for the transfer of a burst of information. No other device can communicate over the channel path during the time a burst is transferred. The burst can consist of a few bytes, a whole block of data, a sequence of blocks with associated control and status information (the block lengths may be zero), or status information that monopolizes the channel path. The facilities of the channel path capable of operating in the burst mode may be shared by a number of concurrently operating I/O devices.

Some channel paths can tolerate an absence of data transfer for about a half minute during a burst-mode operation, such as occurs when a long gap on magnetic tape is read. An equipment malfunction may be indicated when an absence of data transfer exceeds the prescribed limit.

In the byte-multiplex mode, the I/O device stays logically connected to the channel path only for a short interval of time. The facilities of a channel path capable of operating in the byte-multiplex mode may be shared by a number of concurrently operating I/O devices. In this mode, all I/O operations are split into short intervals of time during which only a segment of information is transferred over the channel path. During such an interval, only one device and its associated subchannel are logically connected to the channel path. The intervals associated with the concurrent operation of multiple I/O devices are sequenced in response to demands from the devices. The channel-subsystem facility associated with a subchannel exercises its controls for any one operation only for the time required to transfer a segment of information. The segment can consist of a single byte of data, a few bytes of data, a status report from the device, or a control sequence used for the initiation of a new operation.

Ordinarily, devices with high data-transfer-rate requirements operate with the channel path in the frame-multiplex mode, slower devices operate in the burst mode, and the slowest devices operate in the byte-multiplex mode. Some control units have a manual switch for setting the desired mode of operation.

An I/O operation that occurs on a parallel-I/O-interface type of channel path may occur in either the burst mode or the byte-multiplex mode depending on the facilities provided by the channel path and the I/O device. For improved performance, some channel paths and control units are provided with facilities for high-speed transfer and data streaming. See the System Library publication IBM System/360 and System/370 I/O Interface Channel to Control Unit OEMI, GA22-6974, for a description of those two facilities.

An I/O operation that occurs on a serial-I/O-interface type of channel path may occur in either the frame-multiplex mode or the burst mode. For improved performance, some control
units attaching to the serial-I/O interface provide the capability to provide sense data to the program concurrent with the presentation of unit-check status, if permitted to do so by the program. (See "Concurrent Sense" on page 17-24.)

Depending on the control unit or channel sub-system, access to a device through a subchannel may be restricted to a single channel-path type.

The modes and features described above affect only the protocol used to transfer information over the channel path and the speed of transmission. No effects are observable by CPU or channel programs with respect to the way these programs are executed.

Control Units

A control unit provides the logical capabilities necessary to operate and control an I/O device and adapts the characteristics of each device so that it can respond to the standard form of control provided by the channel subsystem.

Communication between the control unit and the channel subsystem takes place over a channel path. The control unit accepts control signals from the channel subsystem, controls the timing of data transfer over the channel path, and provides indications concerning the status of the device.

The I/O device attached to the control unit may be designed to perform only certain limited operations, or it may perform many different operations. A typical operation is moving a recording medium and recording data. To accomplish its operations, the device needs detailed signal sequences peculiar to its type of device. The control unit decodes the commands received from the channel subsystem, interprets them for the particular type of device, and provides the signal sequence required for the performance of the operation.

A control unit may be housed separately, or it may be physically and logically integrated with the I/O device, the channel subsystem, or a CPU. In the case of most electromechanical devices, a well-defined interface exists between the device and the control unit because of the difference in the type of equipment the control unit and the device require. These electromechanical devices often are of a type where only one device of a group attached to a control unit is required to transfer data at a time (magnetic-tape units or disk-access mechanisms, for example), and the control unit is shared among a number of I/O devices. On the other hand, in some electronic I/O devices, such as the channel-to-channel adapter, the control unit does not have an identity of its own.

From the programmer's point of view, most functions performed by the control unit can be merged with those performed by the I/O device. Therefore, this publication normally makes no specific mention of the control-unit function; the performance of I/O operations is described as if the I/O devices communicated directly with the channel subsystem. Reference is made to the control unit only when emphasizing a function performed by it or when describing how the sharing of the control unit among a number of devices affects the performance of I/O operations.

I/O Devices

An input/output (I/O) device provides external storage, a means of communication between data-processing systems, or a means of communication between a system and its environment. I/O devices include such equipment as magnetic-tape units, direct-access-storage devices (for example, disks), display units, typewriter-keyboard devices, printers, teleprocessing devices, and sensor-based equipment. An I/O device may be physically distinct equipment, or it may share equipment with other I/O devices.

Most types of I/O devices, such as printers, or tape devices, use external media, and these devices are physically distinguishable and identifiable. Other types are solely electronic and do not directly handle physical recording media. The channel-to-channel adapter, for example, provides for data transfer between two channel paths, and the data never reaches a physical recording medium outside main storage. Similarly, communication controllers may handle the transmission of information between the data-processing system and a remote station, and its input and output are signals on a transmission line.

In the simplest case, an I/O device is attached to one control unit and is accessible from one channel path. Switching equipment is available to make some devices accessible from two or more channel paths by switching devices among control
units and by switching control units among
channel paths. Such switching equipment pro-
vides multiple paths by which an I/O device may
be accessed. Multiple channel paths to an I/O
device are provided to improve performance or I/O
availability, or both, within the system. The man-
gement of multiple channel paths to devices is
under the control of the channel subsystem and
the device, but the channel paths may indirectly
be controlled by the program.

I/O Addressing

Four different types of I/O addressing are provided
by the channel subsystem for the necessary
addressing of the various components: channel-
path identifiers, subchannel numbers, device
numbers, and, though not visible to programs,
addresses dependent on the channel-path type.

Channel-Path Identifier

The channel-path identifier (CHPID) is a system-
unique eight-bit value assigned to each installed
channel path of the system. A CHPID is used to
address a channel path. A CHPID is specified by
the second-operand address of RESET CHANNEL
PATH and used to designate the channel path
that is to be reset. The channel paths by which a
device is accessible are identified in the
subchannel-information block (SCHIB), each by its
associated CHPID, when STORE SUBCHANNEL
is executed. The CHPID can also be used in
operator messages when it is necessary to identify
a particular channel path. A system model may
provide as many as 256 channel paths. The
maximum number of channel paths and the
assignment of CHPIDs to channel paths depends
on the system model.

Subchannel Number

A subchannel number is a system-unique 16-bit
value used to address a subchannel. This value
is unique within a channel subsystem. The sub-
channel is addressed by eight I/O instructions:
CANCEL SUBCHANNEL, CLEAR SUBCHANNEL,
HALT SUBCHANNEL, MODIFY SUBCHANNEL,
RESUME SUBCHANNEL, START SUBCHANNEL,
STORE SUBCHANNEL, and TEST SUB-
CHANNEL. All I/O functions relative to a specific
I/O device are specified by the program by desig-
nating a subchannel assigned to the I/O device.

Subchannels are always assigned subchannel
numbers within a single range of contiguous
numbers. The lowest-numbered subchannel is
subchannel 0. The highest-numbered subchannel
of the channel subsystem has a subchannel
number equal to one less than the number of sub-
channels provided. A maximum of 65,536 sub-
channels can be provided. Normally, subchannel
numbers are only used in communication between
the CPU program and the channel subsystem.

Device Number

Each subchannel that has an I/O device assigned
to it also contains a parameter called the device
number. The device number is a 16-bit value that
is assigned as one of the parameters of the sub-
channel at the time the device is assigned to the
subchannel. The device number uniquely identi-
ifies a device to the program.

The device number provides a means to identify a
device, independent of any limitations imposed by
the system model, the configuration, or channel-
path protocols. The device number is used in
communications concerning the device that take
place between the system and the system oper-
ator. For example, the device number is entered
by the system operator to designate the input
device to be used for initial program loading.

Programming Note: The device number is
assigned at device-installation time and may have
any value. However, the user must observe any
restrictions on device-number assignment that
may be required by the control program, support
programs, or the particular control unit or I/O
device.

Device Identifier

A device identifier is an address, not apparent to
the program, that is used by the channel sub-
system to communicate with I/O devices. The
type of device identifier used depends on the spe-
cific channel-path type and the protocols provided.
Each subchannel contains one or more device
identifiers.

For a channel path of the parallel-I/O-interface
type the device identifier is called a device
address and consists of an eight-bit value. For
the ESCON I/O interface, the device identifier con-
sists of a four-bit control-unit address and an
eight-bit device address. For the FICON I/O interface, the device identifier consists of an eight-bit control-unit-image ID and an eight-bit device address. For the FICON-converted I/O interface, the device identifier consists of a four-bit control-unit address and an eight-bit device address.

The device address identifies the particular I/O device (and, on the parallel-I/O interface, the control unit) associated with a subchannel. The device address may identify, for example, a particular magnetic-tape drive, disk-access mechanism, or transmission line. Any number in the range 0-255 can be assigned as a device address.

For further information about the device identifier used with a particular channel-path type, see the appropriate publication for the channel-path type.

### Execution of I/O Operations

I/O operations are initiated and controlled by information with three types of formats: the instruction START SUBCHANNEL, channel-command words (CCWs), and orders. The START SUBCHANNEL instruction is executed by a CPU and is part of the CPU program that supervises the flow of requests for I/O operations from other programs that manage or process the I/O data.

When START SUBCHANNEL is executed, parameters are passed to the target subchannel requesting that the channel subsystem perform a start function with the I/O device associated with the subchannel. The channel subsystem performs the start function by using information at the subchannel, including the information passed during the execution of the START SUBCHANNEL instruction, to find an accessible channel path to the device. Once the device has been selected, the execution of an I/O operation is accomplished by the decoding and execution of a CCW by the channel subsystem and the I/O device. One or more CCWs arranged for sequential execution form a channel program and are executed as one or more I/O operations, respectively. Both instructions and CCWs are fetched from main storage, and their formats are common for all types of I/O devices, although the modifier bits in the command code of a CCW may specify device-dependent conditions for the execution of an operation at the device.

Operations peculiar to a device, such as rewinding tape or positioning the access mechanism on a disk drive, are specified by orders that are decoded and executed by I/O devices. Orders may be transferred to the device as modifier bits in the command code of a control command, may be transferred to the device as data during a control or write operation, or may be made available to the device by other means.

### Start-Function Initiation

CPU programs initiate I/O operations with the instruction START SUBCHANNEL. This instruction passes the contents of an operation-request block (ORB) to the subchannel. The contents of the ORB include the subchannel key, the address of the first CCW to be executed, and a specification of the format of the CCWs. The CCW specifies the command to be executed and the storage area, if any, to be used.

When the ORB contents have been passed to the subchannel, the execution of START SUBCHANNEL is complete. The results of the execution of the instruction are indicated by the condition code set in the program-status word.

When facilities become available, the channel subsystem fetches the first CCW and decodes it according to the format bit specified in the ORB. If the format bit is zero, format-0 CCWs are specified. If the format bit is one, format-1 CCWs are specified. Format-0 and format-1 CCWs contain the same information, but the fields are arranged differently in the format-1 CCW so that 31-bit addresses can be specified directly in the CCW.

### Path Management

If the first CCW passes certain validity tests and does not have the suspend flag specified as one, the channel subsystem attempts device selection by choosing a channel path from the group of channel paths that are available for selection. A control unit that recognizes the device identifier connects itself logically to the channel path and responds to its selection. The channel subsystem sends the command-code part of the CCW over the channel path, and the device responds with a status byte indicating whether the command can be executed. The control unit may logically dis-
connect from the channel path at this time, or it may remain connected to initiate data transfer.

If the attempted selection does not occur as a result of either a busy indication or a path-not-operational condition, the channel subsystem attempts to select the device by an alternate channel path if one is available. When selection has been attempted on all paths available for selection and the busy condition persists, the operation remains pending until a path becomes free. If a path-not-operational condition is detected on one or more of the channel paths on which device selection was attempted, the program is alerted by a subsequent I/O interruption. The I/O interruption occurs either upon execution of the channel program (assuming the device was selected on an alternate channel path) or as a result of the execution being abandoned because path-not-operational conditions were detected on all of the channel paths on which device selection was attempted.

Channel-Program Execution

If the command is initiated at the device and command execution does not require any data to be transferred to or from the device, the device may signal the end of the operation immediately on receipt of the command code. In operations that involve the transfer of data, the subchannel is set up so that the channel subsystem will respond to service requests from the device and assume further control of the operation.

An I/O operation may involve the transfer of data to or from one storage area, designated by a single CCW, or to or from a number of noncontiguous storage areas. In the latter case, generally a list of CCWs is used for the execution of the I/O operation, with each CCW designating a contiguous storage area and the CCWs are coupled by data chaining. Data chaining is specified by a flag in the CCW and causes the channel subsystem to fetch another CCW upon the exhaustion or filling of the storage area designated by the current CCW. The storage area designated by a CCW fetched on data chaining pertains to the I/O operation already in progress at the I/O device, and the I/O device is not notified when a new CCW is fetched.

Provision is made in the CCW format for the programmer to specify that, when the CCW is decoded, the channel subsystem request an I/O interruption as soon as possible, thereby notifying a CPU program that chaining has progressed at least as far as that CCW in the channel program.

To complement dynamic address translation in CPUs, CCW indirect data addressing is provided. A flag in the CCW specifies that an indirect-data-address list is to be used to designate the storage areas for that CCW. Each time the boundary of a block of storage is reached, the list is referenced to determine the next block of storage to be used. The ORB specifies whether the size of each block of storage is 2K bytes or 4K bytes. CCW indirect data addressing permits essentially the same CCW sequences to be used for a program running with dynamic address translation active in the CPU as would be used if the CPU were operating with equivalent contiguous real storage. CCW indirect data addressing permits the program to designate data blocks having absolute storage addresses up to $2^{64} - 1$, independent of whether format-0 or format-1 CCWs have been specified in the ORB.

In general, the execution of an I/O operation or chain of operations involves as many as three levels of participation:

1. Except for effects due to the integration of CPU and channel-subsystem equipment, a CPU is busy for the duration of the execution of START SUBCHANNEL, which lasts until the addressed subchannel has been passed the ORB contents.

2. The subchannel is busy for a new START SUBCHANNEL from the receipt of the ORB contents until the primary interruption condition is cleared at the subchannel.

3. The I/O device is busy from the initiation of the first operation at the device until either the subchannel becomes suspended or the secondary interruption condition is placed at the subchannel. In the case of a suspended subchannel, the device again becomes busy when the execution of the suspended channel program is resumed.

Chapter 13. I/O Overview 13-7
Conclusion of I/O Operations

The conclusion of an I/O operation normally is indicated by two status conditions: channel end and device end. The channel-end condition indicates that the I/O device has received or provided all data associated with the operation and no longer needs channel-subsystem facilities. This condition is called the primary interruption condition, and the channel end in this case is the primary status. Generally, the primary interruption condition is any interruption condition that relates to an I/O operation and that signals the conclusion at the subchannel of the I/O operation or chain of I/O operations.

The device-end signal indicates that the I/O device has concluded execution and is ready to perform another operation. This condition is called the secondary interruption condition, and the device end in this case is the secondary status. Generally, the secondary interruption condition is any interruption condition that relates to an I/O operation and that signals the conclusion at the device of the I/O operation or chain of operations. The secondary interruption condition can occur concurrently with, or later than, the primary interruption condition.

Concurrent with the primary or secondary interruption conditions, both the channel subsystem and the I/O device can provide indications of unusual situations.

The conditions signaling the conclusion of an I/O operation can be brought to the attention of the program by I/O interruptions or, when the CPUs are disabled for I/O interruptions, by programmed interrogation of the channel subsystem. In the former case, these conditions cause storing of the I/O-interruption code, which contains information concerning the interrupting source. In the latter case, the interruption code is stored as a result of the execution of TEST PENDING INTERRUPTION.

When the primary interruption condition is recognized, the channel subsystem attempts to notify the program, by means of an interruption request, that a subchannel contains information describing the conclusion of an I/O operation at the subchannel. The information identifies the last CCW used and may provide its residual byte count, thus describing the extent of main storage used. Both the channel subsystem and the I/O device may provide additional indications of unusual conditions as part of either the primary or the secondary interruption condition. The information contained at the subchannel may be stored by the execution of TEST SUBCHANNEL or the execution of STORE SUBCHANNEL. This information, when stored, is called a subchannel-status word (SCSW).

Facilities are provided for the program to initiate the execution of a chain of I/O operations with a single START SUBCHANNEL instruction. When the current CCW specifies command chaining and no unusual conditions have been detected during the operation, the receipt of the device-end signal causes the channel subsystem to fetch a new CCW. If the CCW passes certain validity tests and the suspend flag is not specified as a one in the new CCW, execution of a new command is initiated at the device. If the CCW fails to pass the validity tests, the new command is not initiated, command chaining is suppressed, and the status associated with the new CCW causes an interruption condition to be generated. If the suspend flag is specified as a one and this value is valid because of a one value in the suspend control, bit 4 of word 1 of the associated ORB, execution of the new command is not initiated, and command chaining is concluded.

Execution of the new command is initiated by the channel subsystem in the same way as in the previous operation. The ending signals occurring at the conclusion of an operation caused by a CCW specifying command chaining are not made available to the program. When another I/O operation is initiated by command chaining, the channel subsystem continues execution of the channel program. If, however, an unusual condition has been detected, command chaining is suppressed, the channel program is terminated, an interruption condition is generated, and the ending signals causing the termination are made available to the program.

The suspend-and-resume function provides the program with control over the execution of a channel program. The initiation of the suspend function is controlled by the setting of the suspend-control bit in the ORB. The suspend function is signaled to the channel subsystem during channel-program execution when the suspend-control bit in the ORB is one and the
suspend flag in the first CCW or in a CCW fetched during command chaining is one.

Suspension occurs when the channel subsystem fetches a CCW with the suspend flag validly (because of a one value of the suspend-control bit in the ORB) specified as one. The command in this CCW is not sent to the I/O device, and the device is signaled that the chain of commands is concluded. A subsequent RESUME SUBCHANNEL instruction informs the channel subsystem that the CCW that caused suspension may have been modified and that the channel subsystem must refetch the CCW and examine the current setting of the suspend flag. If the suspend flag is found to be zero in the CCW, the channel subsystem resumes execution of the chain of commands with the I/O device.

Channel-program execution may be terminated prematurely by CANCEL SUBCHANNEL, HALT SUBCHANNEL or CLEAR SUBCHANNEL. The execution of CANCEL SUBCHANNEL causes the channel subsystem to terminate the start function at the subchannel if the channel program has not been initiated at the device. When the start function is terminated by the execution of CANCEL SUBCHANNEL, the channel subsystem sets condition code 0 in response to the CANCEL SUBCHANNEL instruction. The execution of HALT SUBCHANNEL causes the channel subsystem to issue the halt signal to the I/O device and terminate channel-program execution at the subchannel. When channel-program execution is terminated by the execution of HALT SUBCHANNEL, the program is notified of the termination by means of an I/O-interruption request. The interruption request is generated when the device presents status for the terminated operation. If, however, the halt signal was issued to the device during command chaining after the receipt of device end but before the next command was transferred to the device, the interruption request is generated after the device has been signaled. In the latter case, the device-status field of the SCSW will contain zeros. The execution of CLEAR SUBCHANNEL clears the subchannel of indications of the channel program in execution, causes the channel subsystem to issue the clear signal to the I/O device, and causes the channel subsystem to generate an I/O-interruption request to notify the program of the completion of the clear function.

I/O Interruptions

Conditions causing I/O-interruption requests are asynchronous to activity in CPUs, and more than one condition can occur at the same time. The conditions are preserved at the subchannels until cleared by TEST SUBCHANNEL or CLEAR SUBCHANNEL, or reset by an I/O-system reset.

When an I/O-interruption condition has been recognized by the channel subsystem and indicated at the subchannel, an I/O-interruption request is made pending for the I/O-interruption subclass specified at the subchannel. The I/O-interruption subclass for which the interruption is made pending is under programmed control through the use of MODIFY SUBCHANNEL. A pending I/O interruption may be accepted by any CPU that is enabled for interruptions from its I/O-interruption subclass. Each CPU has eight mask bits, in control register 6, that control the enablement of that CPU for each of the eight I/O-interruption subclasses, with the I/O mask, bit 6 in the PSW, being the master I/O-interruption mask for the CPU.

When an I/O interruption occurs at a CPU, the I/O-interruption code is stored in the I/O-communication area of that CPU, and the I/O-interruption request is cleared. The I/O-interruption code identifies the subchannel for which the interruption was pending. The conditions causing the generation of the interruption request may then be retrieved from the subchannel explicitly by TEST SUBCHANNEL or by STORE SUBCHANNEL.

A pending I/O-interruption request may also be cleared by TEST PENDING INTERRUPTION when the corresponding I/O-interruption subclass is enabled but the PSW has I/O interruptions disabled or by TEST SUBCHANNEL when the CPU is disabled for I/O interruptions from the corresponding I/O-interruption subclass. A pending I/O-interruption request may also be cleared by CLEAR SUBCHANNEL. Both CLEAR SUBCHANNEL and TEST SUBCHANNEL clear the preserved interruption condition at the subchannel as well.

Normally, unless the interruption request is cleared by CLEAR SUBCHANNEL, the program issues TEST SUBCHANNEL to obtain information concerning the execution of the operation.
Chapter 14. I/O Instructions

I/O-Instruction Formats .......................... 14-1
I/O-Instruction Execution ....................... 14-1
    Serialization .................................. 14-1
    Operand Access ................................. 14-1
    Condition Code ................................. 14-2
    Program Exceptions .................. 14-2
Instructions ..................................... 14-2
    CANCEL SUBCHANNEL ....................... 14-4
    CLEAR SUBCHANNEL .......................... 14-5
    HALT SUBCHANNEL ............................ 14-6
    MODIFY SUBCHANNEL .......................... 14-7
    RESET CHANNEL PATH .......................... 14-9
    RESUME SUBCHANNEL ......................... 14-10
    SET ADDRESS LIMIT ......................... 14-11
    SET CHANNEL MONITOR ....................... 14-12
    START SUBCHANNEL ............................ 14-14
    STORE CHANNEL PATH STATUS ............... 14-16
    STORE CHANNEL REPORT WORD .............. 14-17
    STORE SUBCHANNEL ............................ 14-17
    TEST PENDING INTERRUPTION ........-------- 14-18
    TEST SUBCHANNEL .............................. 14-20

All the I/O instructions described here are pro-
vided for the control of channel-subsystem oper-
ations. The I/O instructions are listed in
Figure 14-1 on page 14-3 All of the I/O
instructions are privileged instructions.

Several I/O instructions result in the channel sub-
system being signaled to perform functions asyn-
chronous to the execution of the instructions. The
description of each instruction of this type contains
a section, “Associated Functions,” that summa-
rizes the asynchronous functions.

I/O-Instruction Formats

All I/O instructions use the S format:

<table>
<thead>
<tr>
<th>Op Code</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

The use of the second-operand address and
general registers 1 and 2 (as implied operands)
depends on the I/O instruction Figure 14-1 on
page 14-3 defines which operands are used to
execute each I/O instruction. In addition, detailed
information regarding operand usage appears in
the description of each I/O instruction.

All I/O instructions that reference a subchannel
use the contents of general register 1 as an
implied operand. For these I/O instructions,
general register 1 contains the subsystem-
identification word. The format of the subsystem-
identification word is as follows:

<table>
<thead>
<tr>
<th>0000000000000001</th>
<th>Subchannel Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32 48 63</td>
</tr>
</tbody>
</table>

Bit positions 48-63 of general register 1 contain
the binary number of the subchannel to be used
for the function specified by the instruction. Bit
positions 0-31 of general register 1 are ignored
and bits 32-47 specify the binary number one.

I/O-Instruction Execution

Serialization

The execution of any I/O instruction causes serial-
ization and checkpoint synchronization to occur.
For a definition of the serialization of CPU oper-
ations, see "CPU Serialization" on page 5-102

Operand Access

During the execution of an I/O instruction, the
order in which fields of the operand and fields of
the subchannel, if applicable, are accessed is
unpredictable. It is also unpredictable whether
fetch accesses are made to fields of an operand
or the subchannel, as applicable, when those
fields are not needed to complete the execution of
the I/O instruction. (See "Relation between
Operand Accesses" on page 5-101)
**Condition Code**

During the execution of some I/O instructions, the results of certain tests are used to set one of four condition codes in the PSW. The I/O instructions for which execution can result in the setting of the condition code are listed in Figure 14-1 on page 14-3. The condition code indicates the result of the execution of the I/O instruction. The general meaning of the condition code for I/O instructions is given below; the meaning of the condition code for a specific instruction appears in the description of that instruction.

**Condition Code 0:** Instruction execution produced the expected or most probable result. (See “Deferred Condition Code (CC)” on page 16-8 for a description of conditions that can be encountered subsequent to the presentation of condition code 0 that result in a nonzero deferred condition code.)

**Condition Code 1:** Instruction execution produced the alternate or second-most-probable result, or status conditions were present that may or may not have prevented the expected result.

**Condition Code 2:** Instruction execution was ineffective because the designated subchannel or channel-subsystem facility was busy with a previously initiated function.

**Condition Code 3:** Instruction execution was ineffective because the designated element was not operational or because some condition precluded initiation of the normal function.

In situations where conditions exist that could cause more than one nonzero condition code to be set, the priority of the condition codes is as follows:

Condition code 3 has precedence over condition codes 1 and 2.

Condition code 1 has precedence over condition code 2.

**Program Exceptions**

The program exceptions that the I/O instructions can encounter are access, operand, privileged-operation, and specification exceptions. Figure 14-1 on page 14-3 shows the exceptions that are applicable to each of the I/O instructions. The execution of the instruction is suppressed for privileged-operation, operand, and specification exceptions. Except as indicated otherwise in the section “Special Conditions” for each instruction, the instruction ending for access exceptions is as described in “Recognition of Access Exceptions” on page 6-36.

**Instructions**

The mnemonics, format, and operation codes of the I/O instructions are given in Figure 14-1 on page 14-3. The figure also indicates the conditions that can cause a program interruption and whether the condition code is set.

In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designation for the assembler language are shown with each instruction. For START SUBCHANNEL, for example, SSCH is the mnemonic and D₁(B₁) the operand designation.
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL SUBCHANNEL</td>
<td>XSCH</td>
<td>P</td>
<td>OP $ GS</td>
</tr>
<tr>
<td>CLEAR SUBCHANNEL</td>
<td>CSCH</td>
<td>P</td>
<td>OP $ GS</td>
</tr>
<tr>
<td>HALT SUBCHANNEL</td>
<td>HSCH</td>
<td>P</td>
<td>OP $ GS</td>
</tr>
<tr>
<td>MODIFY SUBCHANNEL</td>
<td>MSCH</td>
<td>P A SP</td>
<td>OP $ GS</td>
</tr>
<tr>
<td>RESET CHANNEL PATH</td>
<td>RCHP</td>
<td>P</td>
<td>OP $ G1</td>
</tr>
<tr>
<td>RESUME SUBCHANNEL</td>
<td>RSCH</td>
<td>P</td>
<td>OP $ GS</td>
</tr>
<tr>
<td>SET ADDRESS LIMIT</td>
<td>SAL</td>
<td>P</td>
<td>OP $ G1</td>
</tr>
<tr>
<td>SET CHANNEL MONITOR</td>
<td>SCHM</td>
<td>P</td>
<td>OP $ GM</td>
</tr>
<tr>
<td>START SUBCHANNEL</td>
<td>SSCH</td>
<td>P A SP</td>
<td>OP $ GS</td>
</tr>
<tr>
<td>STORE CHANNEL PATH STATUS</td>
<td>STCPS</td>
<td>P A SP</td>
<td>$ ST</td>
</tr>
<tr>
<td>STORE CHANNEL REPORT WORD</td>
<td>STCRW</td>
<td>P A SP</td>
<td>$ ST</td>
</tr>
<tr>
<td>STORE SUBCHANNEL</td>
<td>STSCH</td>
<td>P A SP</td>
<td>$ ST</td>
</tr>
<tr>
<td>TEST PENDING INTERRUPTION</td>
<td>TPI</td>
<td>P A SP</td>
<td>$ ST</td>
</tr>
<tr>
<td>TEST SUBCHANNEL</td>
<td>TSCH</td>
<td>P A SP</td>
<td>$ ST</td>
</tr>
</tbody>
</table>

**Explanation:**

- $ Causes serialization and checkpoint synchronization.
- A Access exceptions for logical addresses.
- A1 When the effective address is zero, it is not used to access storage, and no access exceptions can occur, except that access exceptions may occur during access-register translation.
- B2 B2 field designates an access register in the access-register mode.
- C Condition code is set.
- G1 Instruction execution includes the implied use of general register 1 as a parameter.
- GM Instruction execution includes the implied use of multiple general registers.
- GS Instruction execution includes the implied use of general register 1 as the subsystem-identification word.
- P Privileged-operation exception.
- S S instruction format.
- SP Specification exception.
- ST PER storage-alteration event.

*Figure 14-1. Summary of I/O Instructions*
CANCEL SUBCHANNEL

The current start function, if any, is terminated at the designated subchannel if CANCEL SUBCHANNEL is applicable.

General register 1 contains a subsystem-identification word that designates the subchannel for which the current START FUNCTION, if any, is to be terminated.

If the subchannel (1) is not subchannel active, (2) is start pending, resume pending, or suspended, and (3) is performing only the start function, then the start function at the subchannel is terminated, and the subchannel is made no longer start pending, resume pending, or suspended, as appropriate. In addition, internal indications of busy are reset for the subchannel.

Condition code 0 is set to indicate that the actions described above have been taken.

If an invalid ORB field or a no-path-available condition is present for a previously initiated start function and the condition was not reported during the execution of START SUBCHANNEL, condition code 0 may be indicated for CANCEL SUBCHANNEL provided that the subchannel is not yet status pending to report the error condition; if condition code 0 is presented, no subsequent status is generated to indicate the error condition.

Special Conditions

**Condition code 1** is set, and no other action is taken, when the subchannel is status pending with any status.

**Condition code 2** is set, and no other action is taken, when CANCEL SUBCHANNEL is not applicable and the subchannel is not status pending, or when conditions exist that prevent immediate determination of internal conditions for the subchannel and the CPU has determined to end the instruction. CANCEL SUBCHANNEL is not applicable when the subchannel (1) has no function specified, (2) has a function other than the start function alone specified, (3) is not resume pending, is not start pending, and is not suspended, or (4) is subchannel active.

**Condition code 3** is set, and no other action is taken, when the subchannel is not operational for CANCEL SUBCHANNEL. A subchannel is not operational for CANCEL SUBCHANNEL when the subchannel is not provided by the channel subsystem, has no valid device number assigned to it, or is not enabled.

CANCEL SUBCHANNEL can encounter the program exceptions described or listed below.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

**Resulting Condition Code:**

0 Start function canceled
1 Status pending
2 CANCEL SUBCHANNEL not applicable
3 Not operational

**Program Exceptions:**

- Operand
- Privileged operation

**Programming Notes:**

1. The actions taken by CANCEL SUBCHANNEL are completed during the execution of the instruction. If condition code 0 is presented, there is no subsequent I/O interruption resulting from the terminated I/O operation. However, the device may have signaled a busy condition while the canceled operation was start pending. In this case, the device owes a no-longer-busy signal to the channel subsystem. This may result in unsolicited device-end status before the next operation is initiated at the device.

Condition code 2 may be presented to indicate that the CPU has determined to end the instruction because the internal conditions of the specified subchannel cannot be immediately determined. Subsequent execution of STORE SUBCHANNEL may store an SCSW indicating that CANCEL SUBCHANNEL is applicable (i.e., the subchannel is performing the start function alone, is start pending, resume pending, or suspended, and is not subchannel-active.) In such a case, the
program can either recognize that the I/O operation has not been terminated and allow the operation to start or the program can retry CANCEL SUBCHANNEL in an attempt to terminate the start function. However, retrying CANCEL SUBCHANNEL does not guarantee that the instruction will end with condition code 0.

2. Upon the completion of CANCEL SUBCHANNEL with condition code 0, the subchannel is ready to accept a new start function initiated by START SUBCHANNEL.

**CLEAR SUBCHANNEL**

<table>
<thead>
<tr>
<th>CSCH</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B230'</td>
<td>///////////////</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

The designated subchannel is cleared, the current start or halt function, if any, is terminated at the designated subchannel, and the channel subsystem is signaled to asynchronously perform the clear function at the designated subchannel and at the associated device.

General register 1 contains a subsystem-identification word (SID) that designates the subchannel to be cleared.

If a start or halt function is in progress, it is terminated at the subchannel.

The subchannel is made no longer status pending. All activity, as indicated in the activity-control field of the SCSW, is cleared at the subchannel, except that the subchannel is made clear pending. Any functions in progress, as indicated in the function-control field of the SCSW, are cleared at the subchannel, except for the clear function that is to be performed because of the execution of this instruction.

The channel subsystem is signaled to asynchronously perform the clear function. The clear function is summarized below in the section “Associated Functions” and is described in detail in “Clear Function” on page 15-14.

Condition code 0 is set to indicate that the actions described above have been taken.

**Associated Functions**

Subsequent to the execution of CLEAR SUBCHANNEL, the channel subsystem asynchronously performs the clear function. If conditions allow, the channel subsystem chooses a channel path and attempts to issue the clear signal to the device to terminate the I/O operation, if any. The subchannel then becomes status pending. Conditions encountered by the channel subsystem that preclude issuing the clear signal to the device do not prevent the subchannel from becoming status pending (see “Clear Function” on page 15-14).

When the subchannel becomes status pending as a result of performing the clear function, data transfer, if any, with the associated device has been terminated. The SCSW stored when the resulting status is cleared by TEST SUBCHANNEL has the clear-function bit stored as one. If the channel subsystem can determine that the clear signal was issued to the device, the clear-pending bit is stored as zero in the SCSW. Otherwise, the clear-pending bit is stored as one, and other indications are provided that describe in greater detail the condition that was encountered. (See “Interruption-Response Block” on page 16-6)

Measurement data is not accumulated, and device-connect time is not stored in the extended-status word for the subchannel, for a start function that is terminated by CLEAR SUBCHANNEL.

**Special Conditions**

**Condition code 3** is set, and no other action is taken, when the subchannel is not operational for CLEAR SUBCHANNEL. A subchannel is not operational for CLEAR SUBCHANNEL when the subchannel is not provided in the channel subsystem, has no valid device number assigned to it, or is not enabled.

CLEAR SUBCHANNEL can encounter the program exceptions described or listed below.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Function initiated</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
</tr>
</tbody>
</table>
Program Exceptions:

- Operand
- Privileged operation

HALT SUBCHANNEL

The current start function, if any, is terminated at the designated subchannel, and the channel subsystem is signaled to asynchronously perform the halt function at the designated subchannel and at the associated device.

General register 1 contains a subsystem-identification word that designates the subchannel to be halted.

If a start function is in progress, it is terminated at the subchannel.

The subchannel is made halt pending, and the halt function is indicated at the subchannel.

When HALT SUBCHANNEL is executed and the designated subchannel is subchannel-and-device active and status pending with intermediate status, the status-pending indication is eliminated (see the discussion of bits 24, 25, and 28 in “Activity Control (AC)” on page 16-13). The status-pending condition is reestablished as part of the halt function (see the section “Associated Functions” below).

The channel subsystem is signaled to asynchronously perform the halt function. The halt function is summarized below in the section “Associated Functions” and is described in detail in “Halt Function” on page 15-15.

Condition code 0 is set to indicate that the actions described above have been taken.

Associated Functions

Subsequent to the execution of HALT SUBCHANNEL, the channel subsystem asynchronously performs the halt function. If conditions allow, the channel subsystem chooses a channel path and attempts to issue the halt signal to the device to terminate the I/O operation, if any. The subchannel then becomes status pending.

When the subchannel becomes status pending as a result of performing the halt function, data transfer, if any, with the associated device has been terminated. The SCSW stored when the resulting status is cleared by TEST SUBCHANNEL has the halt-function bit stored as one. If the halt signal was issued to the device, the halt-pending bit is stored as zero. Otherwise, the halt-pending bit is stored as one, and other indications are provided that describe in greater detail the condition that was encountered. (See “Interruption-Response Block” on page 16-6 and “Halt Function” on page 15-15).

On some models, path availability is tested as part of the halt function instead of as part of the execution of the instruction. In these models, when no channel path is available for selection, the halt signal is not issued, and the subchannel is made status pending. When the status-pending condition is subsequently cleared by TEST SUBCHANNEL, the halt-pending bit is stored as one in the SCSW.

If a status-pending condition is eliminated during the execution of HALT SUBCHANNEL, then this condition is reestablished along with the other status conditions when the completion of the halt function is indicated to the program.

The halt-pending condition may not be recognized by the channel subsystem if a status-pending condition has been generated. This situation could occur, for example, when alert status is presented or generated while the subchannel is already start pending or resume pending, or when primary status is presented during the attempt to initiate the I/O operation for the first command as specified by the start function or implied by the resume function. If recognition of the status-pending condition by the channel subsystem has occurred logically prior to recognition of the halt-pending condition, the SCSW, when cleared by TEST SUBCHANNEL, has the halt-pending bit stored as one.

If measurement data is being accumulated when a start function is terminated by HALT SUBCHANNEL, the measurement data continues to be accumulated for the subchannel and reflects the extent of subchannel and device usage required, if
any, while performing the currently terminated start function. The measurement data, if any, is accumulated in the measurement block for the subchannel or placed in the extended-status word, as appropriate, when the subchannel becomes status-pending with primary or secondary status. (See “Channel-Subsystem Monitoring” on page 17-1.)

Special Conditions

**Condition code 1** is set, and no other action is taken, when the subchannel is status pending alone or is status pending with any combination of alert, primary, or secondary status.

**Condition code 2** is set, and no other action is taken, when the subchannel is busy for HALT SUBCHANNEL. The subchannel is busy for HALT SUBCHANNEL when a halt function or clear function is already in progress at the subchannel.

**Condition code 3** is set, and no other action is taken, when the subchannel is not operational for HALT SUBCHANNEL. A subchannel is not operational for HALT SUBCHANNEL when the subchannel is not provided in the channel subsystem, has no valid device number assigned to it, or is not enabled. On some models, a subchannel is also not operational for HALT SUBCHANNEL when no channel path is available for selection by the device. (See “Channel-Path Availability” on page 15-13 for a description of channel paths that are available for selection.)

HALT SUBCHANNEL can encounter the program exceptions described or listed below.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Function initiated</td>
</tr>
<tr>
<td>1</td>
<td>Status pending with other than intermediate status</td>
</tr>
<tr>
<td>2</td>
<td>Busy</td>
</tr>
<tr>
<td>3</td>
<td>Not operational</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Operand
- Privileged operation

Program Exceptions: After the execution of HALT SUBCHANNEL, the status-pending condition indicating the completion of the halt function may be delayed for an extended period of time, for example, when the device is a magnetic-tape unit executing a rewind command.

**MODIFY SUBCHANNEL**

<table>
<thead>
<tr>
<th>MSCH</th>
<th>D2(B2)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B232'</td>
<td>B2</td>
<td>D2</td>
</tr>
</tbody>
</table>

The information contained in the subchannel-information block (SCHIB) is placed in the program-modifiable fields at the subchannel. As a result, the program influences, for that subchannel, certain aspects of I/O processing relative to the clear, halt, resume, and start functions and certain I/O support functions.

General register 1 contains a subsystem-identification word (SID) that designates the subchannel that is to be modified as specified by certain fields of the SCHIB. The second-operand address is the logical address of the SCHIB and must be designated on a word boundary; otherwise, a specification exception is recognized.

The channel-subsystem operations that may be influenced due to placement of SCHIB information in the subchannel are:

- I/O processing (E field)
- Interruption processing (interruption parameter and ISC field)
- Path management (D, LPM, and POM fields)
- Monitoring and address-limit checking (measurement-block index, LM, and MM fields)
- Concurrent-sense facility (S field)
- Measurement-block address (MBA)

Bits 0, 1, 6, and 7 of word 1, and bits 0-28 of word 6 of the SCHIB operand must be zeros, and bits 9 and 10 of word 1 must not both be ones. When the extended-I/O-measurement-block facility is installed, bits 26-31 of word 11 must be specified as zeros. When the extended-I/O-measurement-block facility is not installed, bit 29 of word 6 must be specified as zero; otherwise, an operand exception is recognized. When the extended-I/O-measurement-word
facility is not installed, or is installed but not enabled, bit 30 of word 6 must be specified as zero; otherwise, an operand exception is recognized. The remaining fields of the SCHIB are ignored and do not affect the processing of MODIFY SUBCHANNEL. (For further details, see “Subchannel-Information Block” on page 15-1.)

Condition code 0 is set to indicate that the information from the SCHIB has been placed in the program-modifiable fields at the subchannel, except that, when the device-number-valid bit (V) at the designated subchannel is zero, then condition code 0 is set, and the information from the SCHIB is not placed in the program-modifiable fields.

Special Conditions

**Condition code 1** is set, and no other action is taken, when the subchannel is status pending. (See “Status Control (SC)” on page 16-16)

**Condition code 2** is set, and no other action is taken, when a clear, halt, or start function is in progress at the subchannel. (See “Function Control (FC)” on page 16-12)

**Condition code 3** is set, and no other action is taken, when the subchannel is not operational for MODIFY SUBCHANNEL. A subchannel is not operational for MODIFY SUBCHANNEL when the subchannel is not provided in the channel sub-system.

MODIFY SUBCHANNEL can encounter the program exceptions described or listed below.

In word 1 of the SCHIB, bits 0, 1, 6, and 7 must be zeros, and bits 9 and 10 must not both be ones. In word 6 of the SCHIB, bits 0-28 must be zeros. Otherwise an operand exception is recognized.

When the extended-I/O-measurement-block facility is installed, bits 26-31 of word 11 must be specified as zeros; otherwise, an operand exception is recognized.

When the extended-I/O-measurement-word facility is not installed, or is installed but not enabled, bit 30 or word 6 must be specified as zero; otherwise, an operand exception is recognized.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

The execution of MODIFY SUBCHANNEL is suppressed on all addressing and protection exceptions.

**Resulting Condition Code:**

0 Function completed
1 Status pending
2 Busy
3 Not operational

**Program Exceptions:**

- Access (fetch, operand 2)
- Operand
- Privileged operation
- Specification

**Programming Notes:**

1. If a device signals I/O-error alert while the associated subchannel is disabled, the channel subsystem issues the clear signal to the device and discards the I/O-error-alert indication without generating an I/O-interruption condition.

2. If a device presents unsolicited status while the associated subchannel is disabled, that status is discarded by the channel subsystem without generating an I/O-interruption condition. However, if the status presented contains unit check, the channel subsystem issues the clear signal for the associated subchannel and does not generate an I/O-interruption condition. This should be taken into account when the program uses MODIFY SUBCHANNEL to enable a subchannel. For example, the medium on the associated device that was present when the subchannel became disabled may have been replaced, and, therefore, the program should verify the integrity of that medium.

3. It is recommended that the program inspect the contents of the subchannel by subsequently issuing STORE SUBCHANNEL when
MODIFY SUBCHANNEL sets condition code 0. Use of STORE SUBCHANNEL is a method for determining if the designated subchannel was changed or not. Failure to inspect the subchannel following the setting of condition code 0 by MODIFY SUBCHANNEL may result in conditions that the program does not expect to occur.

**RESET CHANNEL PATH**

| Channel-path-reset facility is signaled to perform the channel-path-reset function on the channel path designated by the contents of general register 1. |

The format of general register 1 is as follows:

| 0 32 56 63 | 00000000000000000000000000000000 CHPID |

**Channel-Path Identifier (CHPID):** Bit positions 56-63 of general register 1 contain an unsigned binary integer that designates the channel path on which the channel-path-reset function is to be performed.

Bit positions in general register 1 that are shown as zeros, are reserved and must contain zeros; otherwise, an operand exception is recognized. Bit positions 0-31 of general register 1 are ignored.

If conditions allow, the channel-path-reset facility is signaled to asynchronously perform the channel-path-reset function on the designated channel path. The channel-path-reset function is summarized below in the section “Associated Functions” and is described in detail in “Channel-Path Reset” on page 17-13.

Condition code 0 is set to indicate that the channel-path-reset facility has been signaled.

**Associated Functions**

Subsequent to the execution of RESET CHANNEL PATH, the channel-path-reset facility asynchronously performs the channel-path-reset function. Certain indications are reset at all subchannels that have access to the designated channel path, and the reset signal is issued on that channel path. Any I/O functions in progress at the devices are reset, but only for the channel path on which the reset signal is received. An I/O operation or chain of I/O operations taking place in the multipath mode may be able to continue to be executed on other channel paths in the multipath group, if any. (See "Channel-Path-Reset Function" on page 15-45)

The result of performing the channel-path-reset function on the designated channel path is communicated to the program by means of a channel report (see "Channel Report" on page 17-25).

**Special Conditions**

**Condition code 2** is set, and no other action is taken, when, on some models, the channel-path-reset facility is busy performing the channel-path-reset function for a previous execution of the RESET CHANNEL PATH instruction.

**Condition code 3** is set, and no other action is taken, when, on some models, the designated channel path is not operational for the execution of RESET CHANNEL PATH. On these models, the channel path is not operational for the execution of RESET CHANNEL PATH when the designated channel path is not physically available.

If the channel-path-reset facility is busy and the designated channel path is not physically available, it depends on the model whether condition code 2 or 3 is set.

RESET CHANNEL PATH can encounter the program exceptions described or listed below.

Bit positions 32-55 of general register 1 must contain zeros; otherwise, an operand exception is recognized. Bit positions 0-31 of general register 1 are ignored.

**Resulting Condition Code:**

| 0 1 |
| Function initiated | -- |
2 Busy
3 Not operational

Program Exceptions:

- Operand
- Privileged operation

Programming Notes:

1. To eliminate the possibility of a data-integrity exposure for devices that have the capability of generating unsolicited device-end status, I/O operations in progress with such devices on the channel path for which RESET CHANNEL PATH is to be executed must be terminated by the execution of either HALT SUBCHANNEL or CLEAR SUBCHANNEL. Otherwise, subsequent to receiving the reset signal, the device may present an unsolicited device end that may be interpreted by the channel subsystem as a solicited device end and cause command chaining to occur.

2. If the status-verification facility is being used and RESET CHANNEL PATH is executed without first stopping all ongoing operations associated with the channel path being reset, erroneous device-status-check conditions may be detected.

RESUME SUBCHANNEL

<table>
<thead>
<tr>
<th>RSCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B238'</td>
</tr>
<tr>
<td>[S]</td>
</tr>
</tbody>
</table>

The channel subsystem is signaled to perform the resume function at the designated subchannel.

General register 1 contains a subsystem-identification word that designates the subchannel at which the resume function is to be performed.

The subchannel is made resume pending.

Logically prior to the setting of condition code 0 and only if the subchannel is currently in the suspended state, path-not-operational conditions at the subchannel, if any, are cleared.

The channel subsystem is signaled to asynchronously perform the resume function. The resume function is summarized below in the section ‘Associated Functions’ and is described in detail in ‘Start Function and Resume Function’ on page 15-18.

Condition code 0 is set to indicate that the actions described above have been taken.

Associated Functions

Subsequent to the execution of RESUME SUBCHANNEL, the channel subsystem asynchronously performs the resume function. Except when the subchannel is subchannel active, if the execution of RESUME SUBCHANNEL results in the setting of condition code 0, performance of the resume function causes execution of a currently suspended channel program to be resumed with the associated device, provided that the suspend flag for the current CCW has been set to zero by the program. If the suspend flag remains one, execution of the channel program remains suspended. But, if the subchannel is subchannel active at the time the execution of RESUME SUBCHANNEL results in the setting of condition code 0, then it is unpredictable whether execution of the current program is resumed or whether it is found by the resume function that the subchannel has become suspended in the interim. The subchannel is found to be suspended by the resume function only if the subchannel is status pending with intermediate status when the resume-pending condition is recognized by the channel subsystem. (See ‘Start Function and Resume Function’ on page 15-18.)

Special Conditions

Condition code 1 is set, and no other action is taken, when the subchannel is status pending.

Condition code 2 is set, and no other action is taken, when the resume function is not applicable. The resume function is not applicable when the subchannel (1) has any function other than the start function alone specified, (2) has no function specified, (3) is resume pending, or (4) does not have suspend control specified for the start function in progress.

Condition code 3 is set, and no other action is taken, when the subchannel is not operational for the resume function. A subchannel is not operational for the resume function if the subchannel is not provided in the channel subsystem, has no
valid device number assigned to it, or is not enabled.

RESUME SUBCHANNEL can encounter the program exceptions described or listed below.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

Resulting Condition Code:

0  Function initiated
1  Status pending
2  Function not applicable
3  Not operational

Program Exceptions:

- Operand
- Privileged operation

Programming Notes:

1. When channel-program execution is resumed from the suspended state, the device views the resumption as the beginning of a new chain of commands. When the suspension of channel-program execution occurs and the device requires that certain commands be first or appear only once in a chain of commands (for example, direct-access-storage devices), the program must ensure that the appropriate commands in the proper sequence are fetched by the channel subsystem after channel-program execution is resumed. One way the program can ensure proper sequencing of commands at the device is by allowing the I/O interruption to occur for an intermediate interruption condition due to suspension.

   It is not reliable to notify the program that the subchannel is suspended by using the PCI flag in the CCW that contains the S flag because the PCI I/O interruption may occur before the subchannel is suspended. The SCSW would indicate that an I/O operation is in progress at the subchannel and device in this case.

   The suspend flag of the target CCW should be set to zero before RESUME SUBCHANNEL is executed; otherwise, it is possible that the resume-pending condition may be recognized and the CCW refetched while the suspend flag is still one, in which case the resume-pending condition would be reset, and the execution of the channel program would be suspended. If the suspend flag of the target CCW is set to zero before the execution of RESUME SUBCHANNEL, the channel program is not suspended, provided that the subchannel is not subchannel active at the time the execution of RESUME SUBCHANNEL results in setting of condition code 0. If condition code 0 is set while the subchannel is still subchannel active, it is unpredictable whether the resume-pending condition is recognized by the channel subsystem or whether it is found by the resume function that the subchannel has become suspended in the interim. The subchannel is found to be suspended by the resume function only if the subchannel is status pending with intermediate status at the time the resume-pending condition is recognized. When the subchannel is suspended, the execution of TEST SUBCHANNEL, which clears the intermediate interruption condition, also clears the indication of resume pending.

2. Some models recognize a resume-pending condition only after a CCW having an S flag validly set to one is fetched. Therefore, if a subchannel is resume pending and, during the execution of the channel program, no CCW is fetched having an S flag validly set to one, the subchannel remains resume pending until the primary interruption condition is cleared by TEST SUBCHANNEL.

3. Path availability is not tested during the execution of RESUME SUBCHANNEL. Instead, path availability is tested when the channel subsystem begins performance of the resume function.

4. The contents of the CCW fetched during performance of the resume function may be different from the contents of the same CCW when it was previously fetched and contained an S flag validly set to one.

SET ADDRESS LIMIT

The address-limit-checking facility is signaled to use the specified address as the address-limit
value, and the specified address is passed to the facility. Depending on the model, this instruction may not be provided. When this instruction is not provided, it is checked for operand exception and privileged-operation exception, and then is suppressed.

General register 1 contains the absolute address to be used as the address-limit value. The specified address must be on a 64K-byte boundary and may designate a maximum absolute storage address of 2,147,418,112 (7FFF0000 hex) regardless of whether the CPU is operating in the 24-bit, 31-bit, or 64-bit addressing mode. Bits 0-31 of general register 1 are ignored, and bit 32 must be zero.

General register 1 has the following format:

```
/// 0  Address-Limit Value
0 32 63
```

Associated Functions

The value that is used by the address-limit-checking facility when determining whether to permit or prohibit a data access is called the address-limit value. The initial address-limit value is zero. The initial address-limit value is used by the address-limit-checking facility until the facility recognizes a signal, caused by the execution of SET ADDRESS LIMIT, to use a specified address. The recognition of this specified address as the new address-limit value occurs asynchronously with respect to the execution of SET ADDRESS LIMIT.

If address-limit checking is specified for a sub-channel, then whether the specified address is used by the address-limit-checking facility, when determining whether to permit or prohibit a data access, depends on whether SET ADDRESS LIMIT was executed before, during, or after the execution of START SUBCHANNEL for that sub-channel. If SET ADDRESS LIMIT is executed before START SUBCHANNEL, the specified address is used by the address-limit-checking facility. If SET ADDRESS LIMIT is executed during or after the execution of START SUBCHANNEL, it is unpredictable whether the specified address is used by the address-limit-checking facility for that particular start function. For a description of the manner in which address-limit checking is performed, see "Address-Limit Checking" on page 17-23.

Special Conditions

SET ADDRESS LIMIT can encounter the program exceptions described or listed below.

The address in general register 1 must be designated on a 64K byte boundary, and bit 32 of general register 1 must be zero; otherwise, an operand exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operand
- Privileged operation

**SET CHANNEL MONITOR**

SCHM [S]

```
'B23C' /////////////////
0 16 31
```

Each of the measurement-block-update mode and device-connect-time-measurement mode of the channel subsystem is made either active or inactive, depending on the values of the measurement-mode-control bits in general register 1. If the measurement-mode-control bit for measurement-block update is one, the measurement-block origin and measurement-block key are passed to the channel subsystem.

General register 1 has the following format:

```
/// MBK 00000000000000000000000000000000
0 32 36 62 63
```

**Ignored:** Bit positions 0-31 of general register 1 are ignored.

**Measurement-Block Key (MBK):** Bit positions 32-35 of general register 1 contain the measurement-block key. When bit 62 is one, MBK specifies the access key that is to be used by the channel subsystem when it accesses the measurement-block area and, when the extended-I/O-measurement-block facility is
installed, to access format-1 measurement blocks. Otherwise, MBK is ignored.

**Measurement-Block-Update Control (M):** Bit 62 of general register 1 is the measurement-mode-control bit that controls the measurement-block-update mode. When bit 62 of general register 1 is one and conditions allow, the measurement-block-update facility is signaled to asynchronously make the measurement-block-update mode active. In addition, the measurement-block-origin (MBO) address in general register 2 and the measurement-block key (MBK) in general register 1 are passed to the measurement-block-update facility. The measurement-block origin is used to determine the location of format-0 measurement blocks; the address of format-1 measurement blocks is stored at the subchannel using MODIFY SUBCHANNEL. The measurement-block key is used to access both format-0 and format-1 measurement blocks. The asynchronous functions that are performed by the measurement-block-update facility are summarized below in the section “Associated Functions” and are described in detail in “Channel-Subsystem Monitoring” on page 17-1.

When bit 62 of general register 1 is zero and conditions allow, the measurement-block-update mode is made inactive if it is active or remains inactive if it is inactive. The contents of bit positions 32-35 (MBK) of general register 1 and the contents of general register 2 are ignored.

**Device-Connect-Time-Measurement Control (D):** Bit 63 of general register 1 is the measurement-mode-control bit (D). When bit 63 is one and conditions allow, the device-connect-time-measurement mode is made active if it is inactive or remains active if it is active. When bit 63 is zero and conditions allow, the device-connect-time-measurement mode is made inactive if it is active or remains inactive if it is inactive.

Bit positions 36-61 of general register 1 are reserved and must contain zeros; otherwise, an operand exception is recognized.

General register 2 has the following format:

<table>
<thead>
<tr>
<th>MBO Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

**Measurement-Block-Origin (MBO) Address:** When bit 62 (M) of general register 1 is one, bit positions 0-63 of general register 2 contain the absolute address of the measurement-block origin (MBO), which is the beginning of the measurement-block area. The MBO address is used by the channel subsystem to locate format-0 measurement blocks. The origin of the measurement-block area must be designated on a 32-byte boundary; otherwise, an operand exception is recognized. When bit 62 of general register 1 is zero, the contents of general register 2 are ignored.

If the channel-subsystem timer that is used by the channel-subsystem-monitoring facilities is in the error state, the state is reset. This happens independent of the setting of the two measurement-mode-control bits. (See “Channel-Subsystem Timing” on page 17-2 for a description of the timing facilities.)

**Associated Functions**

When the measurement-block-update facility is signaled (by means of SET CHANNEL MONITOR) to make the measurement-block-update mode active, the functions that are performed by the facility depend on whether or not the mode is already active when the signal is generated.

If the measurement-block-update mode is inactive when the signal is generated, the mode remains inactive until the measurement-block-update facility recognizes the signal. When the measurement-block-update facility recognizes the signal, the measurement-block-update mode is made active, and the MBK that was passed when the signal was generated is used to access all measurement blocks, and the MBO that was passed when the signal was generated is used to determine the address of format-0 measurement blocks.

If the measurement-block-update mode is active when the signal is generated, the mode remains active, and the MBK and MBO associated with the execution of a previous SET CHANNEL MONITOR instruction continue to be used to control the storing of measurement data until the measurement-block-update facility recognizes the signal. When the measurement-block-update facility recognizes the signal, the MBK and MBO associated with that signal are used instead of the
MBK and MBO associated with the execution of a previous SET CHANNEL MONITOR instruction. The SET CHANNEL MONITOR instruction does not affect the measurement-block address used for format-1 measurement blocks, but the MBK associated with the signal becomes the key used to access the measurement block.

In all above cases, the measurement-block-update facility recognizes the signal during, or subsequent to, the execution of the SET CHANNEL MONITOR instruction that caused the signal to be generated and logically prior to the performance of any start function that is initiated by the subsequent execution of START SUBCHANNEL for a subchannel that is enabled for measurement by this facility. If a subchannel that is enabled for measurement by this facility already has a start function in progress when the signal is generated, it is unpredictable when measurement data for that subchannel is stored by using the MBK and MBO associated with that signal.

While the measurement-block-update mode is active, performance measurements are accumulated for subchannels that are enabled for measurement-block update. Measurements for a subchannel are either accumulated in a single 32-byte format-0 measurement block within the measurement-block area, or a 64-byte format-1 measurement block pointed to by the measurement-block address at the subchannel. A subchannel is enabled for the measurement-block-update mode by setting the measurement-block-update-enable bit to one in the SCHIB and then executing the MODIFY SUBCHANNEL instruction for that subchannel. The measurement-block-format-control bit (F) at the subchannel specifies whether a format-0 or format-1 measurement block is stored for a subchannel when the measurement-block-update mode is active and the subchannel is enabled for measurement-block updates. When the F bit is zero, the MBO and MBI are used to determine the address of the measurement block for the subchannel, and a format-0 measurement block is stored. When the F bit is one, the measurement-block-address field at the subchannel contains the address of the measurement block for the subchannel, and a format-1 measurement block is stored. The F bit and measurement-block-address field are modified using the MODIFY SUBCHANNEL instruction.

When the device-connect-time-measurement mode is active, measurements of the length of time that the device is actively communicating with the channel subsystem during the execution of a channel program are accumulated for subchannels that are enabled for device-connect-time measurement. Measurements for a subchannel are provided in the extended-status word ESW of the IRB. A subchannel is enabled for device-connect-time-measurement mode by setting the device-connect-time-measurement-enable bit to one in the SCHIB and then executing MODIFY SUBCHANNEL for that subchannel.

For a more detailed description of the measurement-block-update mode, the format and contents of the measurement block, and the device-connect-time-measurement mode, see "Channel-Subsystem Monitoring" on page 17-1.

Special Conditions

SET CHANNEL MONITOR can encounter the program exceptions described or listed below.

Bits 36-61 of general register 1 must be zeros. When bit 62 (M) of general register 1 is one, the MBO address in general register 2 must be designated on a 32-byte boundary. Otherwise, an operand exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Operand
- Privileged operation

**Programming Note:** When the channel subsystem is initialized, the measurement-block-update and device-connect-time-measurement modes are made inactive.

**START SUBCHANNEL**

<table>
<thead>
<tr>
<th>SSCH</th>
<th>D2 (B2)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B233'</td>
<td>B2</td>
<td>D2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The channel subsystem is signaled to asynchronously perform the start function for the associated device, and the execution parameters that are contained in the designated ORB are placed
at the designated subchannel. (See "Operation-Request Block" on page 15-22.)

General register 1 contains a subsystem-identification word that designates the subchannel to be started. The second-operand address is the logical address of the ORB and must be designated on a word boundary; otherwise, a specification exception is recognized.

The execution parameters contained in the ORB are placed at the subchannel.

When START SUBCHANNEL is executed, the subchannel is status pending with only secondary status, and the extended-status-word-format bit (L) is zero, the status-pending condition is discarded at the subchannel.

The subchannel is made start pending, and the start function is indicated at the subchannel.

Logically prior to the setting of condition code 0, path-not-operational conditions at the subchannel, if any, are cleared.

The channel subsystem is signaled to asynchronously perform the start function. The start function is summarized below in the section “Associated Functions” and is described in detail in “Start Function and Resume Function” on page 15-18.

Condition code 0 is set to indicate that the actions described above have been taken.

Associated Functions

Subsequent to the execution of START SUBCHANNEL, the channel subsystem asynchronously performs the start function.

The contents of the ORB, other than the fields that must contain all zeros, are checked for validity. On some models, the fields of the ORB that must contain zeros are checked asynchronously, instead of during the execution of the instruction. When invalid fields are detected asynchronously, the subchannel becomes status pending with primary, secondary, and alert status and with deferred condition code 1 and program check indicated. (See "Program Check" on page 16-24). In this situation, the I/O operation or chain of I/O operations is not initiated at the device, and the condition is indicated by the start-pending bit being stored as one when the SCSW is cleared by the execution of TEST SUBCHANNEL. (See "Subchannel-Status Word" on page 16-6).

On some models, path availability is tested asynchronously, instead of during the execution of the instruction. When no channel path is available for selection, the subchannel becomes status pending with primary and secondary status and with deferred condition code 3 indicated. The I/O operation or chain of I/O operations is not initiated at the device, and this condition is indicated by the start-pending bit being stored as one when the SCSW is cleared by the execution of TEST SUBCHANNEL.

If conditions allow, a channel path is chosen, and execution of the channel program that is designated in the ORB is initiated. (See “Start Function and Resume Function” on page 15-18).

Special Conditions

Condition code 1 is set, and no other action is taken, when the subchannel is status pending when START SUBCHANNEL is executed. On some models, condition code 1 is not set when the subchannel is status pending with only secondary status; instead, the status-pending condition is discarded.

Condition code 2 is set, and no other action is taken, when a start, halt, or clear function is currently in progress at the subchannel (see “Function Control (FC)” on page 16-12).

Condition code 3 is set, and no other action is taken, when the subchannel is not operational for START SUBCHANNEL. A subchannel is not operational for START SUBCHANNEL if the subchannel is not provided in the channel subsystem, has no valid device number associated with it, or is not enabled.

A subchannel is also not operational for START SUBCHANNEL, on some models, when no channel path is available for selection. On these models, the lack of an available channel path is detected as part of the START SUBCHANNEL execution. On other models, channel-path availability is only tested as part of the asynchronous start function.

START SUBCHANNEL can encounter the program exceptions described or listed below.
In word 1 of the ORB, bits 13 and 25-30 must be zeros, and, in word 2 of the ORB, bit 0 must be zero. Otherwise, on some models, an operand exception is recognized. On other models, an I/O-interruption condition is generated, indicating program check, as part of the asynchronous start function.

START SUBCHANNEL can also encounter the program exceptions listed below.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

The execution of START SUBCHANNEL is suppressed on all addressing and protection exceptions.

Responding Condition Code:

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function initiated</td>
<td>0</td>
</tr>
<tr>
<td>Status pending</td>
<td>1</td>
</tr>
<tr>
<td>Busy</td>
<td>2</td>
</tr>
<tr>
<td>Not operational</td>
<td>3</td>
</tr>
</tbody>
</table>

Program Exceptions:

- Access (fetch, operand 2)
- Operand
- Privileged operation
- Specification

STORE CHANNEL PATH STATUS

Depending on the model, this instruction may not be provided. When this instruction is not provided, it is checked for privileged operation exception and the instruction is suppressed by the machine.

A channel-path-status word of up to 256 bits is stored at the designated location.

The second-operand address is the logical address of the location where the channel-path-
Programming Notes:

1. To ensure a consistent interpretation of channel-path-status-word bits, the program should, prior to the initial use of the area, store zeros at the location where the channel-path-status word is to be stored.

**STORE CHANNEL REPORT WORD**

<table>
<thead>
<tr>
<th>STCRW D2(B2)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B239'</td>
<td>B2</td>
</tr>
</tbody>
</table>

0 16 20 31

A CRW containing information affecting the channel subsystem is stored at the designated location.

The second-operand address is the logical address of the location where the CRW is to be stored and must be designated on a word boundary; otherwise, a specification exception is recognized.

When a malfunction or other condition affecting channel-subsystem operation is recognized, a channel report (consisting of one or more CRWs) describing the condition is made pending for retrieval and analysis by the program. The channel report contains information concerning the identity and state of a facility following the detection of the malfunction or other condition. For a description of the channel report, the CRW, and program-recovery actions related to the channel subsystem, see "Channel-Subsystem Recovery" on page 17-24.

When one or more channel reports are pending, the instruction causes a CRW to be stored at the designated location and condition code 0 to be set. A pending CRW can only be stored by the execution of STORE CHANNEL REPORT WORD and, once stored, is no longer pending. Thus, each pending CRW is presented only once to the program.

When no channel reports are pending in the channel subsystem execution of STORE CHANNEL REPORT WORD causes zeros to be stored at the designated location and condition code 1 to be set.

Special Conditions

STORE CHANNEL REPORT WORD can encounter the program exceptions described or listed below.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

The execution of STORE CHANNEL REPORT WORD is suppressed on all addressing and protection exceptions.

Resulting Condition Code:

0 CRW stored
1 Zeros stored
2 --
3 --

Program Exceptions:

- Access (store, operand 2)
- Privileged operation
- Specification

Programming Notes:

1. CRW overflow conditions may occur if STORE CHANNEL REPORT WORD is not executed to clear pending channel reports. If the overflow condition is encountered, one or more channel-report words have been lost. (See "Channel-Subsystem Recovery" on page 17-24 for details.)

2. A pending CRW can be cleared by any CPU in the configuration executing STORE CHANNEL REPORT WORD, regardless of whether a machine-check interruption has occurred in any CPU.

**STORE SUBCHANNEL**

<table>
<thead>
<tr>
<th>STSCH D2(B2)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B234'</td>
<td>B2</td>
</tr>
</tbody>
</table>

0 16 20 31

Control and status information for the designated subchannel is stored in the designated SCHIB.

General register 1 contains a subsystem-identification word that designates the subchannel for which the information is to be stored. The
second-operand address is the logical address of the SCHIB and must be designated on a word boundary; otherwise, a specification exception is recognized.

When the extended-I/O-measurement-block facility is not installed, the information that is stored in the SCHIB consists of a path-management-control word, a SCSW, and three words of model-dependent information. When the extended-I/O-measurement-block facility is installed, the information that is stored in the SCHIB consists of a path-management-control word, a SCSW, the measurement-block-address field, and one word of model-dependent information. (See “Subchannel-Information Block” on page 15-1)

The execution of STORE SUBCHANNEL does not change any information at the subchannel.

Condition code 0 is set to indicate that control and status information for the designated subchannel has been stored in the SCHIB. When the execution of STORE SUBCHANNEL results in the setting of condition code 0, the information in the SCHIB indicates a consistent state of the subchannel.

**Special Conditions**

*Condition code 3* is set, and no other action is taken, when the designated subchannel is not operational for STORE SUBCHANNEL. A subchannel is not operational for STORE SUBCHANNEL if the subchannel is not provided in the channel subsystem.

STORE SUBCHANNEL can encounter the program exceptions described or listed below.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SCHIB stored</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Not operational</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Access (store, operand 2)
- Operand
- Privileged operation
- Specification

**Programming Notes:**

1. Device status that is stored in the SCSW may include device-busy, control-unit-busy, or control-unit-end indications.

2. The information that is stored in the SCHIB is obtained from the subchannel. The STORE SUBCHANNEL instruction does not cause the channel subsystem to interrogate the addressed device.

3. STORE SUBCHANNEL may be executed at any time to sample conditions existing at the subchannel, without causing any pending status conditions to be cleared.

4. Repeated execution of STORE SUBCHANNEL without an intervening delay (for example, to determine when a subchannel changes state) should be avoided because repeated accesses of the subchannel by the CPU may delay or prohibit access of the subchannel by a channel subsystem to update the subchannel.

**TEST PENDING INTERRUPTION**

<table>
<thead>
<tr>
<th>TPI</th>
<th>Dz (Bz)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'B236'</td>
<td>Bz</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The I/O-interruption code for a pending I/O interruption at a subchannel is stored at the location designated by the second-operand address, and the pending I/O-interruption request is cleared.

The second-operand address, when nonzero, is the logical address of the location where the two-word I/O-interruption code, consisting of words 0 and 1, is to be stored. The second-operand address must be designated on a word boundary; otherwise, a specification exception is recognized.

If the second-operand address is zero, the three-word I/O-interruption code, consisting of words 0-2, is stored at real locations 184-195. In this
case, low-address protection and key-controlled protection do not apply.

In the access-register mode when the second-operand address is zero, it is unpredictable whether access-register translation occurs for access register B₂. If the translation occurs, the resulting address-space-control element is not used; that is, the interruption code still is stored at real locations 184-195.

Pending I/O-interruption requests are accepted only for those I/O-interruption subclasses allowed by the I/O-interruption-subclass mask in control register 6 of the CPU executing the instruction. If no I/O-interruption requests exist that are allowed by control register 6, the I/O-interruption code is not stored, the second-operand location is not modified, and condition code 0 is set.

If a pending I/O-interruption request is accepted, the I/O-interruption code is stored, the pending I/O-interruption request is cleared, and condition code 1 is set. The I/O-interruption code that is stored is the same as would be stored if an I/O interruption had occurred. However, PSWs are not swapped as when an I/O-interruption occurs.

The I/O-interruption code that is stored during the execution of the instruction is defined as follows:

<table>
<thead>
<tr>
<th>Word 0</th>
<th>Subsystem-Identification Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word 1</td>
<td>Interruption Parameter</td>
</tr>
<tr>
<td>Word 2</td>
<td>Interruption-Identification Word</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>31</th>
</tr>
</thead>
</table>

Subsystem-Identification Word (SID): Bits 32-63 of the SID is placed in word 0. See “I/O-Instruction Formats” in Chapter 14.

Interruption Parameter: Word 1 contains a four-byte parameter that was specified by the program and passed to the subchannel in word 0 of the ORB or the PMCW. When a device presents alert status and the interruption parameter was not previously passed to the subchannel by an execution of START SUBCHANNEL or MODIFY SUBCHANNEL, this field contains zeros.

**Interruption-Identification Word:** Word 2, when stored, contains the interruption-identification word, which further identifies the source of the I/O-interruption. Word 2 is stored only when the second-operand address is zero. The interruption-identification word is defined as follows:

```
0 2 5 8 31
```

A bit (A): Bit 0 of the interruption-identification word specifies the type of pending I/O-interruption request that was cleared. When bit 0 is zero, the I/O-interruption request was associated with a subchannel.

I/O-Interruption Subclass (ISC): Bit positions 2-4 of the interruption-identification word contain an unsigned binary integer, in the range 0-7, that specifies the I/O-interruption subclass associated with the subchannel for which the pending I/O-interruption request was cleared.

The remaining bit positions are reserved and stored as zeros.

**Special Conditions**

TEST PENDING INTERRUPTION can encounter the program exceptions described or listed below.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

The execution of TEST PENDING INTERRUPTION is suppressed on all addressing and protection exceptions.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>0</th>
<th>Interruption code not stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interruption code stored</td>
</tr>
<tr>
<td>2</td>
<td>--</td>
</tr>
<tr>
<td>3</td>
<td>--</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Access (store, operand 2, second-operand address nonzero only)
- Privileged operation
- Specification
Programming Notes:

1. TEST PENDING INTERRUPTION should only be executed with a second-operand address of zero when I/O interruptions are masked off. Otherwise, an I/O-interruption code stored by the instruction may be lost if an I/O interruption occurs. The I/O-interruption code that identifies the source of an I/O interruption taken subsequent to TEST PENDING INTERRUPTION is also stored at real locations 184-195, replacing an I/O-interruption code that was stored by the instruction.

2. In the access-register mode when the second-operand address is zero, an access exception is recognized if access-register translation occurs and the access register is in error. This exception can be prevented by making the $B_2$ field zero or by placing 00000000 hex, 00000001 hex, or any other valid contents in the access register.

**TEST SUBCHANNEL**

```
TSCH $D_2 (B_2) \{S\}

'B235' $B_2 $D_2

0 16 20 31
```

Control and status information for the subchannel is stored in the designated IRB.

General register 1 contains a subsystem-identification word that designates the subchannel for which the information is to be stored. The second-operand address is the logical address of the IRB and must be designated on a word boundary; otherwise, a specification exception is recognized.

The information that is stored in the IRB consists of a SCSW, an extended-status word, and an extended-control word. (See “Interruption-Response Block” on page 16-6)

If the subchannel is status pending, the status-pending bit of the status-control field is stored as one. Whether or not the subchannel is status pending has an effect on the functions that are performed when TEST SUBCHANNEL is executed.

When the subchannel is status pending and TEST SUBCHANNEL is executed, information, as described above, is stored in the IRB, followed by the clearing of certain conditions and indications that exist at the subchannel as described in Figure 14-2 on page 14-21. If an I/O-interruption request is pending for the subchannel, the request is cleared. Condition code 0 is set to indicate that these actions have been taken.

When the subchannel is not status pending and TEST SUBCHANNEL is executed, information (as described above) is stored in the IRB, and no conditions or indications are cleared. Condition code 1 is set to indicate that these actions have been taken.

Figure 14-2 on page 14-21 describes which conditions and indications are cleared by TEST SUBCHANNEL when the subchannel is status pending. All other conditions and indications at the subchannel remain unchanged.
### Figure 14-2. Conditions and Indications Cleared at the Subchannel by TEST SUBCHANNEL

**Special Conditions**

*Condition code 3* is set, and no other action is taken, when the subchannel is not operational for TEST SUBCHANNEL. A subchannel is not operational for TEST SUBCHANNEL if the subchannel is not provided, has no valid device number associated with it, or is not enabled.

TEST SUBCHANNEL can encounter the program exceptions described or listed below.

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Alert Status Pdg</th>
<th>Int Status Pdg</th>
<th>Pri Status Pdg</th>
<th>Sec Status Pdg</th>
<th>Status Pdg Alone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Control</td>
<td>C</td>
<td>Nc</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Activity Control</td>
<td>Cp</td>
<td>Nr</td>
<td>Cp</td>
<td>Cp</td>
<td>Cp</td>
</tr>
<tr>
<td>Status Control</td>
<td>Cs</td>
<td>Cs</td>
<td>Cs</td>
<td>Cs</td>
<td>Cs</td>
</tr>
<tr>
<td>N condition</td>
<td>C</td>
<td>Nr</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
```

**Explanation:**

*Note that the rightmost column applies to status pending when it is alone. The other four status-pending conditions result in the clearing actions given. These actions apply both when a single status-pending condition occurs and when a combination of the four status-pending conditions occurs. In the combination case, all the clearing actions of the individual cases apply.*

C Cleared.

Cp The resume-, start-, halt-, clear pending, and suspended conditions are cleared.

Cs The status-pending condition is cleared.

Nc Not changed unless function control indicates the halt function and activity control indicates suspended. If both the halt function and suspended are indicated, conditions are cleared as for status pending alone.

Nr Not changed unless activity control indicates suspended and function control indicates the start function with or without the halt function. If the halt function is indicated, the conditions are cleared as for status pending alone. If only the start function is indicated, the resume-pending condition and the N condition are cleared.

Bits 32-47 of the SID must contain 0001 hex; otherwise, an operand exception is recognized.

The second operand must be designated on a word boundary; otherwise, a specification exception is recognized.

When the execution of TEST SUBCHANNEL is terminated on addressing and protection exceptions, the state of the subchannel is not changed.

**Resulting Condition Code:**

- 0 IRB stored; subchannel status pending
- 1 IRB stored; subchannel not status pending
- 2 --
- 3 Not operational

**Program Exceptions:**

- Access (store, operand 2)
- Operand
- Privileged operation
- Specification

**Programming Notes:**

1. Device status that is stored in the SCSW may include device-busy, control-unit-busy, or control-unit-end indications.

2. The information that is stored in the IRB is obtained from the subchannel. The TEST SUBCHANNEL instruction does not cause the channel subsystem to interrogate the addressed device.

3. When an I/O interruption occurs, it is the result of a status-pending condition at the subchannel, and typically TEST SUBCHANNEL is executed to clear the status. TEST SUBCHANNEL may also be executed at any other time to sample conditions existing at the subchannel.

4. Repeated execution of TEST SUBCHANNEL to determine when a start function has been completed should be avoided because there are conditions under which the completion of the start function may or may not be indicated. For example, if the channel subsystem is holding an interface-control-check (IFCC) condition in abeyance (for any subchannel) because another subchannel is already status pending, and if the start function being tested by TEST SUBCHANNEL has as the only path...
available for selection the channel path with the IFCC condition, then the start function may not be initiated until the status-pending condition in the other subchannel is cleared, allowing the IFCC condition to be indicated at the subchannel to which it applies.

5. Repeated execution of TEST SUBCHANNEL without an intervening delay, for example, to determine when a subchannel changes state, should be avoided because repeated accesses of the subchannel by the CPU may delay or prohibit accessing of the subchannel by the channel subsystem. Execution of TEST SUBCHANNEL by multiple CPUs for the same subchannel at approximately the same time may have the same effect and also should be avoided.

6. The priority of I/O-interruption handling by a CPU can be modified by the execution of TEST SUBCHANNEL. When TEST SUBCHANNEL is executed and the designated subchannel has an I/O-interruption request pending, that I/O-interruption request is cleared, and the SCSW is stored, without regard to any previously established priority. The relative priority of the remaining I/O-interruption requests is unchanged.
Chapter 15. Basic I/O Functions

Subchannel-Information Block

The subchannel-information block (SCHIB) is the operand of the MODIFY SUBCHANNEL and STORE SUBCHANNEL instructions. The two rightmost bits of the SCHIB address are zeros, designating the SCHIB on a word boundary. The SCHIB contains three major fields: the path-management-control word (PMCW), the subchannel-status word (SCSW), and a model-dependent area. When the extended-I/O-measurement-block facility is installed the SCHIB also contains the measurement-block-address field. (Figure 15-1 on page 15-2 shows the format of the PMCW, and Figure 16-2 on page 16-7 shows the format of the SCSW.)

STORE SUBCHANNEL is used to store the current PMCW, the SCSW, model-dependent data, and, when the extended-I/O-measurement-block facility is

Some I/O instructions specify to the channel sub-system that a function is to be performed. Collectively, these functions are referred to as the basic I/O functions. The basic I/O functions are the clear, halt, start, resume, and channel-path-reset functions.

Control of Basic I/O Functions

Information that is present at the subchannel controls how the clear, halt, start, resume, and start functions are performed. This information is communicated to the program in the subchannel-information block during the execution of STORE SUBCHANNEL.
installed, the measurement-block-address field, for
the designated subchannel. MODIFY SUB-
CHANNEL alters certain PMCW fields and, when
the extended-I/O-measurement-block facility is
installed, the measurement-block address in the
subchannel. When the program needs to change
the contents of one or more of the PMCW fields,
the normal procedure is to (1) issue STORE SUB-
CHANNEL to obtain the current contents,
(2) perform the required modifications to the
PMCW field or the measurement-block-address
field in main storage, and (3) issue MODIFY SUB-
CHANNEL to pass the new information to the sub-
channel. The SCHIB has the following format:

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Path-Management-Control Word</td>
</tr>
<tr>
<td>1</td>
<td>Subchannel-Status Word</td>
</tr>
<tr>
<td>2</td>
<td>Model-Dependent Area / Measurement-Block Address</td>
</tr>
<tr>
<td>3</td>
<td>Model-Dependent Area</td>
</tr>
</tbody>
</table>

Path-Management-Control Word
Words 0-6 of the SCHIB contain the path-
management-control word (PMCW). The PMCW
has the format shown in Figure 15-1 when the
subchannel is valid (see "Device Number Valid
(V) on page 15-4). The format of the PMCW is as follows:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-31</td>
<td>Interruption Parameter</td>
</tr>
<tr>
<td>2-4</td>
<td>I/O-Interruption-Subclass Code (ISC)</td>
</tr>
<tr>
<td>8</td>
<td>Enabled (E)</td>
</tr>
</tbody>
</table>

Interruption Parameter: Bit positions 0-31 of
word 0 contain the interruption parameter that is
stored as word 1 of the interruption code. The
interruption parameter can be set to any value by
START SUBCHANNEL and MODIFY SUB-
CHANNEL. The initial value of the interruption
parameter is zero.

I/O-Interruption-Subclass Code (ISC): Bits 2-4
of word 1 are an unsigned binary integer, in the
range 0-7, that corresponds to the bit position of
the I/O-interruption subclass-mask bit in control
register 6 of each CPU in the configuration. The
setting of that mask bit in control register 6 of a
CPU controls the recognition of interruption
requests relating to this subchannel by that CPU
(see "Priority of Interruptions" on page 16-4). The
ISC can be set to any value by MODIFY SUB-
CHANNEL. The initial value of the ISC is zero.

Reserved: Bits 0, 1, and 5-7 of word 1 are
reserved and stored as zeros by STORE SUB-
CHANNEL. Bits 0, 1, 6, and 7 must be zeros
when MODIFY SUBCHANNEL is executed; other-
wise, an operand exception is recognized. Bit 5 of
word 1 is ignored when MODIFY SUBCHANNEL
is executed.

Enabled (E): Bit 8 of word 1, when one, indicates
that the subchannel is enabled for all I/O
functions. When the E bit is zero, status pre-
sented by the device is not made available to the
program, and I/O instructions other than MODIFY
SUBCHANNEL and STORE SUBCHANNEL that
are executed for the designated subchannel cause
condition code 3 to be set. The E bit can be
either zero or one when MODIFY SUBCHANNEL
is executed; initially, all subchannels are not
enabled; IPL causes the IPL I/O device to become
enabled.
Limit Mode (LM): When the address-limit-checking facility is installed, bits 9 and 10 of word 1 define the limit mode (LM) of the subchannel. The limit mode is used by the channel subsystem when address-limit checking is invoked for an I/O operation. (See “Address-Limit Checking” on page 17-23.) Address-limit checking is under the control of the address-limit-checking-control bit that is passed to the subchannel in the operation-request block (ORB) during the execution of START SUBCHANNEL. (See “Address-Limit-Checking Control (A)” on page 15-25.) The definitions of the LM bits, whose values are used during data transfer, are as follows:

<table>
<thead>
<tr>
<th>Bit 9</th>
<th>Bit 10</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Initialized value. No limit checking is performed for this subchannel.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Data address must be equal to or greater than the current address limit.</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Data address must be less than the current address limit.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Bit positions 9 and 10 can contain any of the first three bit combinations shown above when MODIFY SUBCHANNEL is executed. Specification of the reserved bit combination in the operand causes an operand exception to be recognized when MODIFY SUBCHANNEL is executed.

When the address-limit-checking facility is not installed, bits 9-10 of word 1 may be set as described above; however, they are ignored and are not set at the specified subchannel. When bits 9-10 are not specified as described above and MODIFY SUBCHANNEL is executed, an operand exception is recognized.

Measurement-Mode Enable (MM): Bits 11 and 12 of word 1 enable the measurement-block-update mode and the device-connect-time-measurement mode, respectively, of the subchannel. These bits can have any value when MODIFY SUBCHANNEL is executed; initially, neither measurement mode is enabled. The definition of each of these bits is as follows:

<table>
<thead>
<tr>
<th>Bit 11</th>
<th>Measurement-Block-Update Enable:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initialized value. The subchannel is not enabled for measurement-block update. Storing of measurement-block data does not occur.</td>
</tr>
<tr>
<td>1</td>
<td>The subchannel is enabled for measurement-block update. If the measurement-block-update mode is active, measurement data is accumulated in the measurement block at the time channel-program execution is completed or suspended at the subchannel or completed at the device, as appropriate, provided no error conditions described by subchannel logout have been detected. (See “Measurement-Block Update” on page 17-3) If the measurement-block-update mode is inactive, no measurement-block data is stored.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 12</th>
<th>Device-Connect-Time-Measurement Enable:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initialized value. The subchannel is not enabled for device-connect-time measurement. Storing of the device-connect-time interval (DCTI) in the extended-status word (ESW) does not occur.</td>
</tr>
<tr>
<td>1</td>
<td>The subchannel is enabled for device-connect-time measurement. If the device-connect-time-measurement mode is active and timing facilities are provided for the subchannel, the value of the DCTI is stored in the ESW when TEST SUBCHANNEL is executed after channel-program execution is completed or suspended at the subchannel, provided no error conditions described by subchannel logout have been detected. If the device-connect-time-measurement mode is inactive, no measurement values are stored in the ESW.</td>
</tr>
</tbody>
</table>

The meaning of the measurement-mode-enable bits (MM), described above, applies when the timing-facility bit for the subchannel is one. When the timing-facility bit is zero, the effect of the MM bits is changed, as described below under “Timing Facility.” (For more discussion on measurement modes, see “Measurement-Block Update” on page 17-3 and “Device-Connect-Time Measurement” on page 17-10.)

Multipath Mode (D): Bit 13 of word 1, when one, indicates that the subchannel operates in the multipath mode when performing an I/O operation.
or chain of I/O operations. For proper operation in the multipath mode when more than one channel path is available for selection, the associated device must have the dynamic-reconnection feature installed and must be set up for multipath-mode operation. During performance of a start function in the multipath mode, a device is allowed to request service from the channel subsystem over any of the channel paths indicated at the subchannel as being available for selection (see “Logical-Path Mask (LPM)” and “Path-Available Mask (PAM)” on page 15-7). Bit 13, when zero, indicates that the subchannel operates in single-path mode when performing an I/O operation or chain of I/O operations. In the single-path mode, the entire start function is performed by using the channel path on which the first command of the I/O operation or chain of I/O operations was accepted by the device. The D bit can be either zero or one when MODIFY SUBCHANNEL is executed; initially, the subchannel is in the single-path mode.

Timing Facility (T): Bit 14 of word 1, when one, indicates that the channel-subsystem-timing facility is available for the subchannel and is under the control of the two measurement-mode-enable bits (MM) and SET CHANNEL MONITOR. Bit 14, when zero, indicates that the channel-subsystem-timing facility is not available for the subchannel. When bit 14 is zero, the START SUBCHANNEL count is the only measurement data that can be accumulated in the measurement block for the subchannel. Storing of the START SUBCHANNEL count is under the control of bit 11 and SET CHANNEL MONITOR, as described above under “Measurement Mode Enable.” Similarly, if the T bit is zero, no device-connect-time-interval (DCTI) values can be measured for the subchannel. (See “Measurement-Block Update” on page 17-3 and “Device-Connect-Time Measurement” on page 17-10.)

Device Number Valid (V): Bit 15 of word 1, when one, indicates that the device-number field (see below) contains a valid device number and that a device associated with this subchannel may be physically installed. Bit 15, when zero, indicates that the subchannel is not valid, there is no I/O device currently associated with the subchannel, and the contents of all other defined fields of the SCHIB are unpredictable.

Device Number: Bit positions 16-31 of word 1 contain the binary representation of the four-digit hexadecimal device number of the device that is associated with this subchannel. The device number is a system-unique parameter that is assigned to the subchannel and the associated device when the device is installed.

Logical-Path Mask (LPM): Bits 0-7 of word 2 indicate the logical availability of channel paths to the associated device. Each bit of the LPM corresponds one-for-one, by relative bit position, with a CHPID located in an associated byte of words 4 and 5 of the SCHIB. A bit set to one means that the corresponding channel path is logically available; a zero means the corresponding channel path is logically not available. When a channel path is logically not available, the channel subsystem does not use that channel path to initiate performance of any clear, halt, resume, or start function, except when a dedicated allegiance exists for that channel path. When a dedicated allegiance exists at the subchannel for a channel path, the logical availability of the channel path is ignored whenever a clear, halt, resume, or start function is performed. (See “Channel-Path Allegiance” on page 15-11). If the subchannel is idle, the logical availability of the channel path is ignored whenever the control unit initiates a request to present alert status to the channel subsystem. The logical availability of a channel path associated with the subchannel can be changed by setting the corresponding LPM bit in the SCHIB and then issuing MODIFY SUBCHANNEL, or by setting the corresponding LPM bit in the ORB and then issuing START SUBCHANNEL. Initially, each installed channel path is logically available.

Path-Not-Operational Mask (PNOM): Any of bits 8-15 of word 2, when one, indicates that a path-not-operational condition has been recognized on the corresponding channel path. Each bit of the PNOM corresponds one-for-one, by relative bit position, with a CHPID located in an associated byte of words 4 and 5 of the SCHIB. The channel subsystem recognizes a path-not-operational condition when, during an attempted device selection in order to perform a clear, halt, resume, or start function, the device associated with the subchannel appears not operational on a channel path that is operational for the subchannel. When a path-not-operational condition is recognized, the state of the channel path changes from operational for the subchannel to not opera-
tional for the subchannel. A channel path is operational for the subchannel if the associated device appeared operational on that channel path the last time the channel subsystem attempted device selection in order to perform a clear, halt, resume, or start function. A device appears to be operational on a channel path when the device responds to an attempted device selection. A channel path is not operational for the subchannel if the associated device appeared not operational on that channel path the last time the channel subsystem attempted device selection in order to perform a clear, halt, resume, or start function. Any of bits 8-15 of word 2, when zero, indicates that a path-not-operational condition has not been recognized on the corresponding channel path.

Initially, each of the eight possible channel paths associated with each subchannel is considered to be operational, regardless of whether the respective channel paths are installed or available; therefore, unless a path-not-operational condition is recognized during initial program loading, the PMCW, if stored, contains a PNOM of all zeros if stored prior to the execution of a CLEAR SUBCHANNEL, HALT SUBCHANNEL, RESUME SUBCHANNEL, or START SUBCHANNEL instruction.

Programming Note: The PNOM indicates those channel paths for which a path-not-operational condition has not been recognized during the performance of the most recent clear, halt, resume, or start function. That is, the PNOM indicates which of the channel paths associated with the subchannel have made a transition from the operational to the not-operational state for the subchannel during the performance of the most recent clear, halt, resume, or start function. However, the transition of a channel path from the not-operational to the operational state for the subchannel is indicated in the POM. Therefore, the POM must be examined in order to determine whether any of the channel paths that are associated with a designated subchannel are operational for the subchannel.

Furthermore, while performing either a start function or a resume function, the transition of a channel path from the not-operational to the operational state for the subchannel is recognized by the channel subsystem only during the initiation sequence for the first command specified by the start function or implied by the resume function. Therefore, a channel path that is currently not operational for the subchannel can be used by the device associated with the subchannel when reconnecting to the channel subsystem in order to continue command chaining; however, the channel subsystem does not indicate a transition of that channel path from the not-operational to the operational state for the subchannel in the POM.

<table>
<thead>
<tr>
<th>Device State before Selection Attempt</th>
<th>Value of Specified Bit Subsequent to Selection Attempt</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>NOP</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>OP</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>NOP</td>
<td>1 0 1 1</td>
</tr>
</tbody>
</table>

Explanation:

1. Device state as it appears on the corresponding channel path.
2. Prior to the attempted device selection during the performance of either a start function or a resume function while the subchannel is suspended, the channel subsystem clears all existing path-not-operational conditions, if any, at the designated subchannel.
3. The N bit (bit 15 of word 0 of the SCSW) is indicated to the program and the N condition is cleared at the subchannel when TEST SUBCHANNEL is executed the next time the subchannel is status pending for other than intermediate status alone provided that it is not also suspended.

NOP The device is not operational on the corresponding channel path.

OP The device is operational on the corresponding channel path.

Figure 15-2. Resulting POM, PNOM, and N-Bit Values Subsequent to Selection Attempt

Last-Path-Used Mask (LPUM): Bits 16-23 of word 2 indicate the channel path that was last used for communicating or transferring information between the channel subsystem and the device. Each bit of the LPUM corresponds one-for-one, by relative bit position, with a CHPID located in an associated byte of words 4 and 5 of the SCHIB. Each bit of the LPUM is stored as zero, except for the bit that corresponds to the channel path last used, whenever one of the following occurs:
1. The first command of a start or resume function is accepted by the device (see “Activity Control (AC)” on page 16-13).

2. The device and channel subsystem are actively communicating when the suspend function is performed for the channel program in execution.

3. Status has been accepted from the device and is recognized as an interruption condition, or a condition has been recognized that suppresses command chaining (see “Interruption Conditions” on page 16-2).

4. An interface-control-check condition has been recognized (see “Interface-Control Check” on page 16-28), and no subchannel-logoff information is currently present in the subchannel.

The LPUM field of the PMCW contains the most recent setting. The initial value of the LPUM is zero.

Path-Installed Mask (PIM): Bits 24-31 of word 2 indicate which of the channel paths 0-7 to the I/O device are physically installed. The PIM indicates the validity of the channel-path identifiers (see below) for those channel paths that are physically installed. Each bit of the PIM corresponds one-for-one, by relative bit position, with a CHPID located in an associated byte of words 4 and 5 of the SCHIB. A PIM bit stored as one indicates that the corresponding channel path is installed. A PIM bit stored as zero indicates that the corresponding channel path is not installed. The PIM always reflects the full complement of installed paths to the device, regardless of how the system is configured. Therefore, some of the channel paths indicated in the PIM may not be physically available in that configuration, as indicated by the bit settings in the path-available mask (see below). The initial value of the PIM indicates all the physically installed channel paths to the device.

Measurement-Block Index (MBI): Bits 0-15 of word 3 form an index value used by the measurement-block-update facility when the measurement-block-update mode is active (see “SET CHANNEL MONITOR” on page 14-13) and the subchannel is enabled for the mode (see “Measurement-Mode Enable (MM)” on page 15-3). When the measurement-block index is used, five zero bits are appended on the right, and the result is added to the measurement-block-origin address designated by SET CHANNEL MONITOR. The calculated address, called the measurement-block address, designates the beginning of a 32-byte storage area where measurement data is stored. (See “Measurement Block” on page 17-3.) The MBI can contain any value when MODIFY SUBCHANNEL is executed; the initial value is zero.

Path-Operational Mask (POM): Bits 16-23 of word 3 indicate the last known operational state of the device on the corresponding channel paths. Each bit of the POM corresponds one-for-one, by relative bit position, with a CHPID located in an associated byte of words 4 and 5 of the SCHIB. If the associated device appeared operational on a channel path the last time the channel subsystem attempted device selection in order to perform a clear, halt, resume, or start function, then the channel path is operational for the subchannel, and the bit corresponding to the channel path in the POM is one. A device appears to be operational on a channel path when the device responds to an attempted device selection. A channel path is also operational for the subchannel if MODIFY SUBCHANNEL is executed and the bit corresponding to that channel path in the POM is specified as one.

If the associated device appeared not operational on a channel path the last time the channel subsystem attempted device selection in order to perform a clear, halt, resume, or start function, then the channel path is not operational for the subchannel, and the bit corresponding to the channel path in the POM is zero. A channel path is also not operational for the subchannel if MODIFY SUBCHANNEL is executed and the bit corresponding to that channel path in the POM is specified as zero.

If the device associated with the subchannel appears not operational on a channel path that is operational for the subchannel during an attempted device selection in order to perform a clear, halt, resume, or start function, then the channel subsystem recognizes a path-not-operational condition. If an SCSW is subsequently stored, then bit 15 of word 0 is one, indicating the path-not-operational condition. When a path-not-operational condition is recognized, the state of the channel path changes from operational for the subchannel to not operational for the subchannel.
When the channel path is not operational for the subchannel, a path-not-operational condition cannot be recognized. Moreover, a channel path that is not operational for the subchannel may be available for selection; if the channel subsystem chooses that channel path while performing a path-management operation, and if, during the attempted device selection, the device appears to be operational again on that channel path, then the state of the channel path changes from not operational for the subchannel to operational for the subchannel.

The POM can contain any value when MODIFY SUBCHANNEL is executed. Initially, each of the eight possible channel paths associated with each subchannel is considered to be operational, regardless of whether the respective channel paths are installed or available; therefore, unless a path-not-operational condition is recognized during initial program loading, the PMCW, if stored, contains a POM of all ones if stored prior to the execution of a CLEAR SUBCHANNEL, HALT SUBCHANNEL, RESUME SUBCHANNEL, or START SUBCHANNEL instruction.

Path-Available Mask (PAM): Bits 24-31 of word 3 indicate the physical availability of installed channel paths. Each bit of the PAM corresponds one-for-one, by relative bit position, with a CHPID located in an associated byte of words 4 and 5 of the SCHIB. A PAM bit of one indicates that the corresponding channel path is physically available for use in accessing the device. A PAM bit of zero indicates the channel path is not physically available for use in accessing the device. When a channel path is not physically available, it may, depending on the model and the extent of failure, be used during performance of the reset-channel-path function. A channel path that is physically available may become not physically available as a result of reconfiguring the system, or this may occur as a result of the performance of the channel-path-reset function. The initial value of the PAM reflects the set of channel paths by which the I/O device is physically accessible at the time of initialization.

Note: The change in the availability of a channel path affects all subchannels having access to that channel path. Whenever the setting of a PAM bit is referred to in conjunction with the availability status of a channel path, for brevity, reference is made in this chapter to a single PAM bit instead of to the respective PAM bits in all of the affected subchannels.

Channel-Path Identifiers (CHPIDs): Words 4 and 5 contain eight one-byte channel-path identifiers corresponding to channel paths 0-7 of the PIM. A CHPID is valid if the corresponding PIM bit is one. Each valid CHPID contains the identifier of a physical channel path to a control unit by which the associated I/O device may be accessed. A unique CHPID is assigned to each physical channel path in the system.

Different devices that are accessible by the same physical channel path have, in their respective subchannels, the same CHPID value. The CHPID value may, however, appear in each subchannel in different locations in the CHPID fields 0-7.

Subchannels that share an identical set of channel paths have the same corresponding PIM bits set to ones. The channel-path identifiers (CHPIDs) for these channel paths are the same and occupy the same respective locations in each SCHIB.

Reserved: Bits 0-28 of word 6 are reserved and are stored as zeros by STORE SUBCHANNEL. They must be zeros when MODIFY SUBCHANNEL is executed; otherwise, an operand exception may be recognized.

Measurement Block Format Control (F): when the extended-measurement-block facility is installed, bit 29 of word 6 specifies the format of the measurement block to be stored when the subchannel is enabled for the measurement-block-update mode, and measurement-block-update mode is active. The bit can contain any value when MODIFY SUBCHANNEL is executed. The initial value is zero. The definition of the bit is as follows:
If the extended-measurement-block facility is not installed, bit 29 of word 6 of the SCHIB operand must be zero when MODIFY SUBCHANNEL is executed; otherwise, an operand exception is recognized.

**Extended Measurement Word Mode Enable (X):** When the extended-measurement-word facility is installed and enabled, bit 30 of word 6 enables the extended-measurement-word mode for the subchannel. Initially, the extended-measurement-word mode is not enabled. The definition of the bit is as follows:

- **Bit 30 Extended-Measurement-Word-Enable:**
  - 0: Initialized value. The subchannel is not enabled for extended-measurement-word mode. Storing of the extended-measurement word does not occur.
  - 1: The subchannel is enabled for extended-measurement-word mode. Measurement data is stored in the extended-measurement word at the time channel-program execution is completed or suspended at the subchannel or completed at the device, as appropriate, provided no error conditions described by subchannel logout have been detected.

If the extended-measurement-word facility is not installed, or is installed but is not enabled, bit 30 of word 6 of the SCHIB operand must be zero when MODIFY SUBCHANNEL is executed; otherwise, an operand exception is recognized.

**Concurrent Sense (S):** Bit 31 of word 6, when one, indicates that the subchannel is in the concurrent-sense mode. When the subchannel is in concurrent-sense mode, whenever the subchannel becomes status pending with alert status, and the status byte accepted from the device contains the unit-check indication, then the channel subsystem may attempt to retrieve sense information from the associated device and place that sense information in the extended-control word.

If the concurrent-sense facility is not installed, bit 31 of word 6 of the SCHIB operand must be zero when MODIFY SUBCHANNEL is executed; otherwise, an operand exception is recognized.

**Subchannel-Status Word**
Words 7-9 of the SCHIB contain a copy of the SCSW. The format of the SCSW is described in "Subchannel-Status Word" on page 16-6. The SCSW is stored by the execution of either STORE SUBCHANNEL or TEST SUBCHANNEL (see "STORE SUBCHANNEL" on page 14-17 and "TEST SUBCHANNEL" on page 14-20).

**Model-Dependent Area/Measurement Block Address**
When the extended-I/O-measurement-block facility is not installed, words 10-12 of the SCHIB contain model-dependent information.

When the extended-I/O-measurement-block facility is installed, words 10-11 are defined as the measurement-block-address field. Word 12 contains model-dependent information.

When (1) the measurement-block-update mode is active (see "SET CHANNEL MONITOR" on page 14-12), (2) the subchannel is enabled for the mode (see "Measurement-Mode Enable (MM)" on page 15-3), and (3) the format-1-measurement block is specified (see "Measurement Block Format Control (F)" on page 15-7) at the subchannel, the measurement-block-address field contains the absolute storage address of the measurement block used by the measurement-block-update facility. The measurement-block address designates the beginning of a 64-byte storage area and must be designated on 64-byte boundary. The initial value of the measurement block address is zero.
Summary of Modifiable Fields

Figure 15-3 on page 15-9 lists the initial settings for fields in a subchannel whose device-number-valid bit is one and indicates what modifies the fields.

All of the PMCW fields contain meaningful information when STORE SUBCHANNEL is executed and the designated subchannel is idle. Subchannel fields that the channel subsystem does not modify contain valid information whenever STORE SUBCHANNEL is executed, provided that the device-number-valid bit is one. The validity of the subchannel fields that are modifiable by the channel subsystem depends on the state of the subchannel at the time STORE SUBCHANNEL is executed.

<table>
<thead>
<tr>
<th>Subchannel Field</th>
<th>Initial Value</th>
<th>Program Modifies by Executing</th>
<th>Modified by Channel Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interruption parameter</td>
<td>Zeros</td>
<td>MSCN,SSCH</td>
<td>No</td>
</tr>
<tr>
<td>I/O-interruption-subclass code</td>
<td>Zeros</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Enabled (E)</td>
<td>Zero</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Limit mode (LM)</td>
<td>Zeros</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Measurement mode (MM)</td>
<td>Zeros</td>
<td>MSCN</td>
<td>Yes</td>
</tr>
<tr>
<td>Multipath mode (Q)</td>
<td>Zero</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Timing facility (T)</td>
<td>Installed value</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Device number valid (V)</td>
<td>Installed value</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Device number</td>
<td>Installed value</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Logical-path mask (LPM)</td>
<td>Path-installed-mask value</td>
<td>MSCN,SSCH</td>
<td>No</td>
</tr>
<tr>
<td>Path-not-operational mask (PNOM)</td>
<td>Zeros</td>
<td>CSCH,SSCH,RSCH</td>
<td>Yes</td>
</tr>
<tr>
<td>Last-path-used mask (LPUM)</td>
<td>Zeros</td>
<td>CSCH</td>
<td>Yes</td>
</tr>
<tr>
<td>Path-installed mask (PIM)</td>
<td>Installed value</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Measurement-block index (MBI)</td>
<td>Zeros</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Path-operational mask (POM)</td>
<td>Ones</td>
<td>CSCH,MSCH,RSCH</td>
<td>Yes</td>
</tr>
<tr>
<td>Path-available mask (PAM)</td>
<td>Installed values</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>Channel-path ID 0-7 (CHPID)</td>
<td>Installed value</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Concurrent sense (S)</td>
<td>Zero</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Subchannel-status word (SCSW)</td>
<td>Zero</td>
<td>TSCH</td>
<td>Yes</td>
</tr>
<tr>
<td>Model-dependent area</td>
<td>*</td>
<td>None</td>
<td>*</td>
</tr>
<tr>
<td>Measurement-block-format control</td>
<td>Zero</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Extended-measurement-word enable</td>
<td>Zero</td>
<td>MSCN</td>
<td>No</td>
</tr>
<tr>
<td>Measurement-block Address</td>
<td>Zeros</td>
<td>MSCN</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 15-3 (Part 1 of 2). Modification of Subchannel Fields
**Explanation:**

1. Model-dependent.
   These fields are not meaningful if the subchannel is not valid. Initialization of a subchannel is performed when I/O-system reset occurs. (See the section “I/O-System Reset” in Chapter 17, “I/O Support Functions.”) One or more of the installed-value parameters that are unmodifiable by the program may be set when the subchannel is idle. In this case, all the program-modifiable fields are set to their initialized values, and the program is notified of such a change by a channel report. (See the section “Channel-Report Word” in Chapter 17, “I/O Support Functions.”)

2. Subchannel fields that are not normally modifiable by the channel subsystem may be modified as a result of dynamic configuration changes or as a result of external actions. When this occurs, the program is notified of the change by a channel report that is made pending at the time of the change.

3. When any of the following error conditions associated with the measurement-block-update mode is detected, the measurement-block-update mode is disabled by the channel subsystem (bit 11 of word 1 of the SCHIB is zero) in the affected subchannel. The device-connect-time-measurement-enable bit (bit 12 of word 1 of the SCHIB) is never modified by the channel subsystem.
   - Measurement program check
   - Measurement protection check
   - Measurement data check
   - Measurement key check

4. This information is entered when the channel-subsystem configuration is established.

5. The mask is modified by the resume function only when the subchannel is in the suspended state at the time RESUME SUBCHANNEL is executed.

6. The channel subsystem may modify the PAM to reflect changes in the system configuration caused by partitioning or unpartitioning channel paths because of reconfiguration or permanent failure of part of the I/O system.

7. The limit mode bits are modified by MSCH only when the instruction is executed when the address-limit-checking facility is installed and the CPC is operating in basic mode.

---

*Figure 15-3 (Part 2 of 2). Modification of Subchannel Fields*
Programming Notes:

1. System performance may be degraded if the LPM is not used to make channel paths for which a path-not-operational condition has been indicated in the PNOM logically not available.

2. If, during the performance of a start function, a channel path becomes not physically available because a channel-path failure has been recognized, continued performance of the start function may be precluded. That is, the program may or may not be notified, and the subchannel may remain in the subchannel-and-device-active state until cleared by the performance of the clear function.

3. If the same MBI is placed in more than one subchannel by the program, the channel-subsystem-monitoring facility updates the same locations with measurement data relating to more than one subchannel. In this case, the values stored in the measurement data are unpredictable. (See “Measurement-Block Update” on page 17-3)

4. Modification of the I/O configuration (reconfiguration) may be accomplished in various ways depending on the model. If the reconfiguration procedure affects the physical availability of a channel path, then any change in availability can be detected by executing STORE SUBCHANNEL for a subchannel that has access to the channel path and by subsequently examining the PAM bits of the SCHIB.

5. The definitions of the PNOM, POM, and N bit are such that a path-not-operational condition is reported to the program only the first time the condition is detected by the channel subsystem after the corresponding POM bit is set to one.

   For example, if the POM bit for every channel path available for selection is one and the device appears not operational on all corresponding channel paths while the channel subsystem is attempting to initiate a start function at the device, the channel subsystem makes the subchannel status pending, with deferred condition code 3 and with the N bit stored as one. The PNOM in the SCHIB indicates the channel path or channel paths that appeared not operational, for which the corresponding POM bits have been set to zeros. The next START SUBCHANNEL causes the channel subsystem to again attempt device selection by choosing a channel path from among all of the channel paths that are available for selection. If device selection is not successful and all channel paths available for selection have again been chosen, deferred condition code 3 is set, but the N bit in the SCSW is zero. The POM contains zeros in at least those bit positions that correspond to the channel paths that are available for selection. (See “Channel-Path Availability” on page 15-13 for a description of the term “available for selection.”) When the N bit in the SCSW is zero, the PNOM is also zero.

6. If the program is to detect path-not-operational conditions, the PNOM should be inspected following the execution of TEST SUBCHANNEL (which results in the setting of condition code zero and the valid storing of the N bit as one) and preceding the performance of another start, resume, halt, or clear function at the subchannel.

Channel-Path Allegiance

The channel subsystem establishes allegiance conditions between subchannels and channel paths. The kind of allegiance established at a subchannel for a channel path or set of channel paths depends upon the state of the subchannel, the device, and the information, if any, transferred between the channel subsystem and device. The way in which path management is handled during the performance of a clear, halt, resume, or start function is determined by the kind of allegiance, if any, currently recognized between a subchannel and a channel path.

Performing the clear function at a subchannel clears any currently existing allegiance condition in the subchannel for all channel paths.

Performing the reset-channel-path function clears all currently existing allegiances for that channel path in all subchannels.

When a channel path becomes not physically available, all internal indications of prior allegiance conditions are cleared in all subchannels having access to the designated channel path.
Working Allegiance

A subchannel has a working allegiance for a channel path when the subchannel becomes device active on that channel path. Once a working allegiance is established, the channel subsystem maintains the working allegiance at the subchannel for the channel path until either the subchannel is no longer device active or a dedicated allegiance is recognized, whichever occurs earlier. Unless a dedicated allegiance is recognized, a working allegiance for a channel path is extended to the set of channel paths that are available for selection if the device is specified to be operating in the multipath mode (that is, the multipath-mode bit is stored as one in the SCHIB). Otherwise, the working allegiance remains only for that channel path over which the start function was initiated.

Once a working allegiance is established for a channel path or set of channel paths, the working allegiance is not changed until the subchannel is no longer device active or until a dedicated allegiance is established. If the subchannel is operating in the single-path mode, a working allegiance is maintained only for a single path.

While a working allegiance exists at a subchannel, an active allegiance can occur only for a channel path for which the working allegiance is being maintained, unless the device is specified as operating in the multipath mode. When the device is specified as operating in the multipath mode, an active allegiance may also occur for a channel path that is not available for selection if the presentation of status by the device on that channel path causes an alert interruption condition to be recognized.

A working allegiance is cleared in any subchannel having access to a channel path if the channel path becomes not physically available.

Active Allegiance

A subchannel has an active allegiance established for a channel path no later than when active communication has been initiated on that channel path with an I/O device. The subchannel can have an active allegiance to only one channel path at a time. While the subchannel has an active allegiance for a channel path, the channel subsystem does not actively communicate with that device on any other channel path. When the channel subsystem accepts a no-longer-busy indication from the device that does not cause an interruption condition, this status does not constitute the initiation of active communication. An active allegiance at a subchannel for a channel path is terminated when the channel subsystem is no longer actively communicating with the I/O device on that channel path.

A working allegiance can become an active allegiance.

Dedicated Allegiance

If a channel path is physically available (that is, if the corresponding PAM bit is one), a dedicated allegiance may be recognized for that channel path. If a channel path is not physically available, a dedicated allegiance cannot be recognized for the corresponding channel path. The channel subsystem establishes a dedicated allegiance at the subchannel for a channel path when (1) the subchannel becomes status pending with alert status, and device status containing the unit-check indication is present but (2) concurrent-sense information is not present at the subchannel. A dedicated allegiance is maintained until the subchannel is no longer start pending (unless it becomes suspended) or resume pending following performance of the next start function, clear function, or channel-path-reset function or the next resume function if applicable. If the subchannel becomes suspended, the dedicated allegiance remains until the resume function is initiated and the subchannel is no longer resume pending. Unless a clear or channel-path-reset function is performed, the subchannel establishes a working allegiance when the dedicated allegiance ends. This occurs when the subchannel becomes device active. While a dedicated allegiance exists at a subchannel for a channel path, only that channel path is available for selection until the dedicated-allegiance condition is cleared.

A dedicated allegiance can become an active allegiance. While a dedicated allegiance exists, an active allegiance can only occur for the same channel path.

A currently existing dedicated allegiance is cleared at any subchannel having access to a channel path when the channel path becomes not phys-
ically available or whenever the device appears not operational on the channel path for which the dedicated allegiance exists.

Channel-Path Availability

When a channel path is not physically available, the channel subsystem does not use the channel path to perform any of the basic I/O functions except, in some cases, the channel-path-reset function and does not respond to any control-unit-initiated requests on that same channel path. If a channel path is not physically available, the condition is indicated by the corresponding path-available-mask PAM bit being zero when STORE SUBCHANNEL is executed (see “Path-Available Mask (PAM)” on page 15-7). Furthermore, if the channel path is not physically available for the subchannel designated by STORE SUBCHANNEL, then it is not physically available for any subchannel that has a device which is accessible by that channel path.

Unless a dedicated allegiance exists at a subchannel for the channel path, a channel path becomes available for selection if it is logically available and physically available, as indicated by the bits in the LPM and PAM corresponding to the channel path being stored as ones when STORE SUBCHANNEL is executed. If a dedicated allegiance exists at a subchannel for the channel path, only that channel path is available for selection, and the setting of the corresponding LPM bit is ignored. If the channel path is currently being used and a dedicated allegiance exists at the subchannel for the channel path, selection of the device is delayed until the channel path is no longer being used.

The availability status of the eight logical paths to the associated device described in Figure 15-4 is determined by the hierarchical arrangement of the corresponding bit values contained in the PIM, PAM, and LPM and by existing conditions, if any, recognized by the channel subsystem.

<table>
<thead>
<tr>
<th>Value of Bit 'n'</th>
<th>Channel-Path Condition</th>
<th>Channel-Path State</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIM</td>
<td>PAM</td>
<td>LPM</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Explanation:
- Bit value is not meaningful.
1. If the channel path is recognized as being used in active communication with a device, the channel-path condition is described as active. Otherwise, its condition is described as inactive.
2. A PAM bit cannot have the value one when the corresponding PIM bit has the value zero.
3. If a dedicated allegiance exists to the channel path at the subchannel, the state of the bit is ignored, and the channel path is considered to be available for selection.
4. The channel path may appear to be active when a channel-path-terminal condition has been recognized.

X Condition is not meaningful.

Figure 15-4. Path Condition and Path-Availability Status for PIM, PAM, and LPM Values

Control-Unit Type

In “Clear Function” on page 15-14, “Halt Function” on page 15-15, and “Start Function and Resume Function” on page 15-18, reference is made to type-1, type-2, and type-3 control units. For a description of these control-unit types, see the System Library publication IBM System/360 and System/370 I/O Interface Channel to Control Unit OEMI, GA22-6974. For the purposes of this definition, all control units attaching to a serial-I/O interface are considered type-2 control units.
Clear Function

Subsequent to the execution of CLEAR SUB-CHANNEL, the channel subsystem performs the clear function. Performance of the clear function consists in (1) performing a path-management operation, (2) modifying fields at the subchannel, (3) issuing the clear signal to the associated device, and (4) causing the subchannel to be made status pending, indicating the completion of the clear function.

Clear-Function Path Management

A path-management operation is performed as part of the clear function in order to examine channel-path conditions for the associated subchannel and to attempt to choose an available channel path on which the clear signal can be issued to the associated device.

Channel-path conditions are examined in the following order:

1. If the channel subsystem is actively communicating or attempting to establish active communication with the device to be signaled, the channel path that is in use is chosen.

2. If the channel subsystem is in the process of accepting a no-longer-busy indication (which will not cause an interruption condition to be recognized) from the device to be signaled, and the associated subchannel has no allegiance to any channel path, the channel path that is in use is chosen.

3. If the associated subchannel has a dedicated allegiance for a channel path, that channel path is chosen.

4. If the associated subchannel has a working allegiance for one or more channel paths, one of those channel paths is chosen.

5. If the associated subchannel has no allegiance for any channel path, if a last-used channel path is indicated, and if that channel path is available for selection, that channel path is chosen. If that channel path is not available for selection, either no channel path is chosen or a channel path is chosen from the set of channel paths, if any, that are available for selection (as though no last-used channel path were indicated).

6. If the associated subchannel has no allegiance for any channel path, if no last-used channel path is indicated, and if there exist one or more channel paths that are available for selection, one of those channel paths is chosen.

If none of the channel-path conditions listed above apply, no channel path is chosen.

For item 4, for item 5 under the specified conditions, and for item 6, the channel subsystem chooses a channel path from a set of channel paths. In these cases, the channel subsystem may attempt to choose a channel path, provided that the following conditions do not apply:

1. A channel-path-terminal condition exists for the channel path.

2. Another subchannel has an active allegiance for the channel path.

3. The device to be signaled is attached to a type-1 control unit, and the subchannel for another device attached to the same control unit has an allegiance to the same channel path, unless the allegiance is a working allegiance and primary status has been accepted by that subchannel.

4. The device to be signaled is attached to a type-3 control unit, and the subchannel for another device attached to the same control unit has a dedicated allegiance to the same channel path.

Clear-Function Subchannel Modification

Path-management-control indications at the subchannel are modified during performance of the clear function. Effectively, this modification occurs after the attempt to choose a channel path, but prior to the attempt to select the device to issue the clear signal. The path-management-control indications that are modified are as follows:

1. The state of all eight possible channel paths at the subchannel is set to operational for the subchannel.

2. The last-path-used indication is reset to indicate no last-used channel path.

3. Path-not-operational conditions, if any, are reset.
Clear-Function Signaling and Completion

Subsequent to the attempt to choose a channel path and the modification of the path-management-control fields, the channel subsystem, if conditions allow, attempts to select the device to issue the clear signal. (See "Clear Signal" on page 17-12.) Conditions associated with the subchannel and the chosen channel path, if any, affect (1) whether an attempt is made to issue the clear signal, and (2) whether the attempt to issue the clear signal is successful. Independent of these conditions, the subchannel is subsequently set status pending, and the performance of the clear function is complete. These conditions and their effect on the clear function are described as follows:

No Attempt Is Made to Issue the Clear Signal:
The channel subsystem does not attempt to issue the clear signal to the device if any of the following conditions exist:

1. No channel path was chosen. (See "Clear-Function Path Management" on page 15-14.)
2. The chosen channel path is no longer available for selection.
3. A channel-path-terminal condition exists for the chosen channel path.
4. The chosen channel path is currently being used to actively communicate with a different device.
5. The device to be signaled is attached to a type-1 control unit, and the subchannel for another device attached to the same control unit has an allegiance to the same channel path, unless the allegiance is a working allegiance and primary status has been accepted by that subchannel.
6. The device to be signaled is attached to a type-3 control unit, and the subchannel for another device attached to the same control unit has a dedicated allegiance to the same channel path.

If any of the conditions above exist, the subchannel remains clear pending and is set status pending, and the performance of the clear function is complete.

The Attempt to Issue the Clear Signal Is Not Successful:
When the channel subsystem attempts to issue the clear signal to the device, the attempt may not be successful because of the following conditions:

1. The control unit or device signals a busy condition when the channel subsystem attempts to select the device to issue the clear signal.
2. A path-not-operational condition is recognized when the channel subsystem attempts to select the device to issue the clear signal.
3. An error condition is encountered when the channel subsystem attempts to issue the clear signal.

If any of the conditions above exists and the channel subsystem either determines that the attempt to issue the clear signal was not successful or cannot determine whether the attempt was successful, the subchannel remains clear pending and is set status pending, and the performance of the clear function is complete.

The Attempt to Issue the Clear Signal Is Successful:
When the channel subsystem determines that the attempt to issue the clear signal was successful, the subchannel is no longer clear pending and is set status pending, and the performance of the clear function is complete. When the subchannel becomes status pending, the I/O operation, if any, with the associated device has been terminated.

Programming Note: Subsequent to the performance of the clear function, any nonzero status, except control unit end alone, that is presented to the channel subsystem by the device is passed to the program as unsolicited alert status. Unsolicited status consisting of control unit end alone or zero status is not presented to the program.

Halt Function

Subsequent to the execution of HALT SUBCHANNEL, the channel subsystem performs the halt function. Performance of the halt function consists of (1) performing a path-management operation, (2) issuing the halt signal to the associated device, and (3) causing the subchannel to be made status pending, indicating the completion of the halt function.
Halt-Function Path Management

A path-management operation is performed as part of the halt function to examine channel-path conditions for the associated subchannel and to attempt to choose a channel path on which the halt signal can be issued to the associated device.

Channel-path conditions are examined in the following order:

1. If the channel subsystem is actively communicating or attempting to establish active communication with the device to be signaled, the channel path that is in use is chosen.

2. If the channel subsystem is in the process of accepting a no-longer-busy indication (which will not cause an interruption condition to be recognized) from the device to be signaled, and the associated subchannel has no allegiance to any channel path, the channel path that is in use is chosen.

3. If the associated subchannel has a dedicated allegiance for a channel path, that channel path is chosen.

4. If the associated subchannel has a working allegiance for one or more channel paths, one of those channel paths is chosen.

5. If the associated subchannel has no allegiance for any channel path, if a last-used channel path is indicated, and if that channel path is available for selection, that channel path is chosen. If that channel path is not available for selection, either no channel path is chosen or a channel path is chosen from the set of channel paths, if any, that are available for selection (as though no last-used channel path were indicated).

6. If the associated subchannel has no allegiance for any channel path, if no last-used channel path is indicated, and if there exist one or more channel paths that are available for selection, one of those channel paths is chosen.

If none of the channel-path conditions listed above apply, no channel path is chosen.

For item 4, for item 5 under the specified conditions, and for item 6, the channel subsystem chooses a channel path from a set of channel paths. In these cases, the channel subsystem may attempt to choose a channel path for which the following conditions do not apply:

1. A channel-path-terminal condition exists for the channel path.

2. Another subchannel has an active allegiance for the channel path.

3. The device to be signaled is attached to a type-1 control unit, and the subchannel for another device attached to the same control unit has an allegiance to the same channel path, unless the allegiance is a working allegiance and primary status has been accepted by that subchannel.

4. The device to be signaled is attached to a type-3 control unit, and the subchannel for another device attached to the same control unit has a dedicated allegiance to the same channel path.

Halt-Function Signaling and Completion

Subsequent to the attempt to choose a channel path, the channel subsystem, if conditions allow, attempts to select the device to issue the halt signal. (See “Halt Signal” on page 17-12.) Conditions associated with the subchannel and the chosen channel path, if any, affect (1) whether an attempt is made to issue the halt signal, (2) whether the attempt to issue the halt signal is successful, and (3) whether the subchannel is made status pending to complete the halt function. These conditions and their effect on the halt function are described as follows:

No Attempt Is Made to Issue the Halt Signal: The channel subsystem does not attempt to issue the halt signal to the device if any of the following conditions exist:

1. No channel path was chosen. (See “Halt-Function Path Management.”)

2. The chosen channel path is no longer available for selection.

3. A channel-path-terminal condition exists for the chosen channel path.

4. The associated subchannel is status pending with other than intermediate status alone.

5. The device to be signaled is attached to a type-1 control unit, and the subchannel for
another device attached to the same control unit has an allegiance to the same channel path, unless the allegiance is a working allegiance and primary status has been accepted by that subchannel.

6. The device to be signaled is attached to a type-3 control unit, and the subchannel for another device attached to the same control unit has a dedicated allegiance to the same channel path.

If the conditions described in items 3 on page 15-16, 5 on page 15-16 or 6 exist, the associated subchannel remains halt pending until those conditions no longer exist. When the conditions no longer exist (for the channel-path-terminal condition, when the condition no longer exists as a result of executing RESET CHANNEL PATH), the channel subsystem attempts to issue the halt signal to the device.

If any of the remaining conditions above exist, the subchannel remains halt pending and is set status pending, and the halt function is complete.

The Attempt to Issue the Halt Signal Is Not Successful: When the channel subsystem attempts to issue the halt signal to the device, the attempt may not be successful because of the following conditions:

1. The control unit or device signals a busy condition when the channel subsystem attempts to select the device to issue the halt signal.
2. A path-not-operational condition is recognized when the channel subsystem attempts to select the device to issue the halt signal.
3. An error condition is encountered when the channel subsystem attempts to issue the halt signal.

If the control unit or device signals a busy condition (item 1), the subchannel remains halt pending until the internal indication of busy is reset. When this event occurs, the channel subsystem again attempts to issue the halt signal to the device.

If any of the remaining conditions above exist and the channel subsystem either determines that the attempt to issue the halt signal was not successful or cannot determine whether the attempt was successful, then the subchannel remains halt pending and is set status pending, and the halt function is complete.

The Attempt to Issue the Halt Signal Is Successful: When the channel subsystem determines that the attempt to issue the halt signal was successful and ending status, if appropriate, has been received at the subchannel, the subchannel is no longer halt pending and is set status pending, and the halt function is complete. When the subchannel becomes status pending, the I/O operation, if any, with the associated device has been terminated. The conditions that affect the receipt of ending status at the subchannel, and the effect of the halt signal at the device are described in the following discussion.

When the subchannel is subchannel-and-device active or only device active during the performance of the halt function, the state continues until the subchannel is made status pending because (1) the device has provided ending status or (2) the channel subsystem has determined that ending status is unavailable. When the subchannel is idle, start pending, start pending and resume pending, suspended, or suspended and resume pending, or when the halt signal is issued during command chaining after the receipt of device end but before the next command is transferred to the device, no operation is in progress at the device, and therefore no status is generated by the device as a result of receiving the halt signal. When the subchannel is neither subchannel active, nor status pending with intermediate status, and no errors are detected during the attempt to issue the halt signal to the device, an interruption condition indicating status pending alone is generated after the halt signal is issued.

The effect of the halt signal at the device depends partially on the type of device and its state. The effect of the halt signal on a device that is not active or that is performing a mechanical operation in which data is not transferred across the channel path, such as rewinding tape or positioning a disk-access mechanism, depends upon the control-unit or device model. If the device is performing a type of operation that is unpredictable in duration or in which data is transferred across the channel path, the control unit interprets the signal as one to terminate the operation. Pending status conditions at the device are not reset. When the control unit recognizes the halt signal, it immediately ceases all communication with the channel subsystem until it has reached the normal ending point. The control unit then requests selection by
the channel subsystem to present any generated status.

If the subchannel is involved in the data-transfer portion of an I/O operation, data transfer is terminated during the performance of the halt function, and the device is logically disconnected from the channel path. If the halt function is addressed to a subchannel performing a chain of I/O operations and the device has already provided channel end for the current I/O operation, the channel subsystem causes the device to be disconnected and command chaining or command retry to be suppressed. If the subchannel is performing a chain of I/O operations with the device and the halt signal is issued during command chaining at a point after the receipt of device end for the previous I/O operation but before the next command is transferred to the device, the subchannel is made status pending with primary and secondary status immediately after the halt signal is issued. The device-status field of the SCSW contains zeros in this case. If the halt function is addressed to a subchannel that is start pending and the halt-pending condition is recognized before initiation of the start function, initiation of the start function is not attempted, and the subchannel becomes status pending after the device has been signaled.

When the subchannel is not performing an I/O operation with the associated device, the device is selected, and an attempt is made to issue the halt signal as the device responds. If the subchannel is in the device-active state, the subchannel does not become status pending until it receives the device-end status from the halted device. If the subchannel is neither subchannel-and-device active nor device active, the subchannel becomes status pending immediately after selecting the device and issuing the halt signal. The SCSW for the latter case has the status-pending bit set to one (see "Status-Pending (Bit 31)" on page 16-18).

The termination of an I/O operation by performing the halt function may result in two distinct interruption conditions.

The first interruption condition occurs when the device generates the channel-end condition. The channel subsystem handles this condition as it would any other interruption condition from the device, except that the command address in the associated SCSW designates the point at which the I/O operation is terminated, and the subchannel-status bits may reflect unusual conditions that were detected. If the halt signal was issued before all data designated for the operation had been transferred, incorrect length is indicated, subject to the control of the SLI flag in the current CCW. The value in the count field of the associated SCSW is unpredictable.

The second interruption condition occurs if device-end status was not presented with the channel-end interruption condition. In this situation, the subchannel-key, command-address, and count fields of the associated SCSW are not meaningful.

When HALT SUBCHANNEL terminates an I/O operation, the method of termination differs from that used upon exhaustion of count or upon detection of programming errors to the extent that termination by HALT SUBCHANNEL is not contingent on the receipt of a service request from the associated device.

Programming Notes:

1. When, after an operation is terminated by HALT SUBCHANNEL, the subchannel is status pending with primary, primary and secondary, or secondary status, the extent of data transferred as described by the count field is unpredictable.

2. When the path that is chosen by the path-management operation has a channel-path-terminal condition associated with it, the halt function remains pending until the condition no longer exists. Until the condition is cleared, the associated subchannel cannot be used to perform I/O operations, even if other channel paths become available for selection. CLEAR SUBCHANNEL can be executed to terminate the halt-pending condition and make the subchannel usable.

Start Function and Resume Function

Subsequent to the execution of START SUBCHANNEL and RESUME SUBCHANNEL, the channel subsystem performs the start and resume functions, respectively, to initiate an I/O operation with the associated device. Performance of a
A path-management operation is performed by the channel subsystem during the performance of either a start or a resume function to choose an available channel path that can be used for device selection to initiate an I/O operation with that device. The actions taken are as follows:

1. If the subchannel is currently start pending and device active, the start function remains pending at the subchannel until the secondary status for the previous start function has been accepted from the associated device and the subchannel is made start pending alone. When the status is accepted and does not describe an alert interruption condition, the subchannel is not made status pending, and the performance of the pending start function is subsequently initiated. If the status describes an alert interruption condition, the subchannel becomes status pending with secondary and alert status, the pending start function is not initiated, deferred condition code 1 is set, and the start-pending bit remains one. If the subchannel is currently start pending alone, the performance of the start function is initiated as described below.

2. If a dedicated allegiance exists at the subchannel for a channel path, the channel subsystem chooses that path for device selection. If a busy condition is encountered while attempting to select the device and a dedicated allegiance exists at the subchannel, the start function remains pending until the internal indication of busy is reset for that channel path. When the internal indication of busy is reset, the performance of the pending start function is initiated on that channel path.

3. If no channel path is available for selection and no dedicated allegiance exists in the subchannel for a channel path, a channel path is not chosen.

4. If all channel paths that are available for selection have been tried and one or more of them are being used to actively communicate with other devices, or, alternatively, if the channel subsystem has encountered either a control-unit-busy or a device-busy condition on one or more of those channel paths, or a combination of those conditions on one or more of those channel paths, the start function remains pending at the subchannel until a channel path, control unit, or device, as appropriate, becomes available.

5. If (1) the start function is to be initiated on a channel path with a device attached to a type-1 control unit and (2) no other device is attached to the same control unit whose subchannel has either a dedicated allegiance to the same channel path or a working allegiance to the same channel path where primary status has not been received for that subchannel, then that channel path is chosen if it is available for selection; otherwise, that channel path is not chosen. If, however, another channel path to the device is available for selection and no allegiances exist as described above, that channel path is chosen.

6. If the device is attached to a type-3 control unit, and if at least one other device is attached to the same control unit whose subchannel has a dedicated allegiance to the same channel path, another channel path that is available for selection may be chosen, or the start function remains pending until the dedicated allegiance for the other device is cleared.

7. If a channel path has been chosen and a busy indication is received during device selection...
to initiate the execution of the first command of a pending channel program, the channel path over which the busy indication is received is not used again for that device or control unit (depending on the device-busy or control-unit-busy indication received) until the internal indication of busy is reset.

8. If, during an attempt to select the device in order to initiate the execution of the first command specified for the start or implied for the resume function (as described in action 7 on page 15-19), the channel subsystem receives a busy indication, it performs one of the following actions:

a. If the device is specified to be operating in the multipath mode and the busy indication received is device busy, then the start or resume function remains pending until the internal indication of busy is reset. (See “Multipath Mode (D)” on page 15-3)

b. If the device is specified to be operating in the multipath mode and the busy indication received is control unit busy, or if the device is specified to be operating in the single-path mode, the channel subsystem attempts selection of the device by choosing an alternate channel path that is available for selection and continues the path-management operation until either the start or the resume function is initiated or selection of the device has been attempted on all channel paths that are available for selection. If the start or resume function has not been initiated by the channel subsystem after all channel paths available for selection have been chosen, the start or resume function remains pending until the internal indication of busy is reset.

c. If the subchannel has a dedicated allegiance, then action 2 on page 15-19 applies.

9. When, during the selection attempt to transfer the first command, the device appears not operational and the corresponding channel path is operational for the subchannel, a path-not-operational condition is recognized, and the state of the channel path changes at the subchannel from operational for the subchannel to not operational for the subchannel (see “Path-Not-Operational Mask (PNOM)” on page 15-4). The path-not-operational conditions at the subchannel, if any, are preserved until the subchannel next becomes clear pending, start pending, or resume pending (if the subchannel was suspended), at which time the path-not-operational conditions are cleared. If, however, the corresponding channel path is not operational for the subchannel, a path-not-operational condition is not recognized. When the device appears not operational during the selection attempt to transfer the first command on a channel path that is available for selection, one of the following actions occurs:

a. If a dedicated allegiance exists for that channel path, then it is the only channel path that is available for selection; therefore, further attempts to initiate the start or resume function are abandoned, and an interruption condition is recognized.

b. If no dedicated allegiance exists and there are alternate channel paths available for selection that have not been tried, one of those channel paths is chosen to attempt device selection and transfer the first command.

c. If no dedicated allegiance exists, no alternate channel paths are available for selection that have not been tried, and the device has appeared operational on at least one of the channel paths that were tried, the start or resume function remains pending at the subchannel until a channel path, a control unit, or the device, as appropriate, becomes available.

d. If no dedicated allegiance exists, no alternate channel paths are available for selection that have not been tried, and the device has appeared not operational on all channel paths that were tried, further attempts to initiate the start or resume function are abandoned, and an interruption condition is recognized.

10. When the subchannel is active and an I/O operation is to be initiated with a device, all device selections occur according to the LPUM indication if the multipath mode is not specified at the subchannel. For example, if command chaining is specified, the channel subsystem transfers the first and all subse-
sequent commands describing a chain of I/O operations over the same channel path.

**Execution of I/O Operations**

After a channel path is chosen, the channel subsystem, if conditions allow, initiates the execution of an I/O operation with the associated device. Execution of additional I/O operations may follow the initiation and execution of the first I/O operation. The channel subsystem can execute seven commands: write, read, read backward, control, sense, sense ID, and transfer in channel. Each command, except transfer in channel, initiates a corresponding I/O operation. Except for periods when channel-program execution is suspended at the subchannel (see "Suspension of Channel-Program Execution" on page 15-38), the subchannel is active from the acceptance of the first command until the primary interruption condition is recognized at the subchannel. If the primary interruption condition is recognized before the acceptance of the first command, the subchannel does not become active. Normally, the primary interruption condition is caused by the channel-end signal or, in the case of command chaining, the channel-end signal for the last CCW of the chain. (See "Primary Interruption Condition" on page 16-4). The device is active until the secondary interruption condition is recognized at the subchannel. Normally, the secondary interruption condition is caused by the device-end signal or, in the case of command chaining, the device-end signal for the last CCW of the chain. (See "Secondary Interruption Condition" on page 16-4)

**Programming Notes:**

In the single-path mode, all transfers of commands, data, and status for the I/O operation or chain of I/O operations occur on the channel path over which the first command was transferred to the device.

When the device has the dynamic-reconnection feature installed, an I/O operation or chain of I/O operations may be performed in the multipath mode. To operate in the multipath mode, MODIFY SUBCHANNEL must have been previously executed for the subchannel with bit 13 of word 1 of the SCHIB specified as one. (See "Multipath Mode (D)" on page 15-3). In addition, the device must be set up for the multipath mode by the execution of certain model-dependent commands appropriate to that type of device. The general procedures for handling multipath-mode operations are as follows:

1. **Setup**

   a. A set-multipath-mode type of command must be successfully executed by the device on each channel path that is to be a member of the multipath group being set up; otherwise, the multipath mode of operation may give unpredictable results at the subchannel. If, for any reason, one or more physically available channel paths to the device are not included in the multipath group, these channel paths must not be available for selection while the subchannel is operating in the multipath mode. A channel path can be made not available for selection by having the corresponding LPM bit set to zero either in the SCHIB prior to the execution of MODIFY SUBCHANNEL or in the ORB prior to the execution of START SUBCHANNEL.

   b. When a set-multipath-mode type of command is transferred to a device, only a single channel path must be logically available in order to avoid alternate channel-path selection for the execution of that start function; otherwise, device-busy conditions may be detected by the channel subsystem on more than one channel path, which may cause unpredictable results for subsequent multipath-mode operations. This type of setup procedure should be used whenever the membership of a multipath group is changed.

2. **Leaving the Multipath Mode**

To leave the multipath mode and continue processing in the single-path mode, either of the following two procedures may be used:

   a. A disband-multipath-mode type of command may be executed for any channel path of the multipath group. This command must be followed by either (1) the execution of MODIFY SUBCHANNEL with bit 13 of word 1 of the SCHIB specified as zero, or (2) the specification of only a single channel path as logically available in the LPM. A start function must not be performed at a subchannel operating in the multipath mode.
with multiple channel paths available for selection while the device is operating in single-path mode; otherwise, unpredictable results may occur at the subchannel for that function or subsequent start functions.

b. A resign-multipath-mode type of command is executed on each channel path of the multipath group (the reverse of the setup described in item 1 on page 15-21). This command must be followed by either (1) the execution of MODIFY SUBCHANNEL with bit 13 of word 1 of the SCHIB specified as zero, or (2) the specification of only a single channel path as logically available in the LPM. No start function may be performed at a subchannel operating in the multipath mode with multiple channel paths available for selection while the device is operating in single-path mode; otherwise, unpredictable results may occur at the subchannel for that or subsequent start functions.

### Blocking of Data

Data recorded by an I/O device is divided into blocks. The length of a block depends on the device; for example, a block can be a card, a line of printing, or the information recorded between two consecutive gaps on magnetic tape.

The maximum amount of information that can be transferred in one I/O operation is one block. An I/O operation is terminated when the associated main-storage area is exhausted or the end of the block is reached, whichever occurs first. For some operations, such as writing on a magnetic-tape unit or at an inquiry station, blocks are not defined, and the amount of information transferred is controlled only by the program.

### Operation-Request Block

The operation-request block (ORB) is the operand of START SUBCHANNEL. The ORB specifies the parameters to be used in controlling that particular start function. These parameters include the interruption parameter, the subchannel key, the address of the first CCW, operation-control bits, numbers controlling priority, and a specification of the logical availability of channel paths.

The contents of the ORB are placed at the designated subchannel during the execution of START SUBCHANNEL, prior to the setting of condition code 0. If the execution will result in a nonzero condition code, the contents of the ORB are not placed at the designated subchannel.

The two rightmost bits of the ORB address must be zeros, placing the ORB on a word boundary; otherwise, a specification exception is recognized. The format of the ORB is as follows:

<table>
<thead>
<tr>
<th>Word</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interruption Parameter</td>
</tr>
<tr>
<td>1</td>
<td>Key S C M Y</td>
</tr>
<tr>
<td>2</td>
<td>LPM</td>
</tr>
<tr>
<td>3</td>
<td>CSS Priority</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The fields in the ORB are defined as follows:

- **Interruption Parameter:** Bits 0-31 of word 0 are preserved unmodified in the subchannel until replaced by a subsequent START SUBCHANNEL or MODIFY SUBCHANNEL instruction. These bits are placed in word 1 of the interruption code when an I/O interruption occurs and when an interruption request is cleared by the execution of TEST PENDING INTERRUPTION.

- **Subchannel Key:** Bits 0-3 of word 1 form the subchannel key for all fetching of CCWs, IDAWs, and output data and for the storing of input data associated with the start function initiated by START SUBCHANNEL. This key is matched with a storage key during these storage references. For details, see the section “Key-Controlled Protection” on page 3-9.

- **Suspend Control (S):** Bit 4 of word 1 controls the performance of the suspend function for the channel program designated in the ORB. The setting of the S bit applies to all CCWs of the channel program designated by the ORB (see “Commands and Flags” on page 15-40). When bit 4 is one, suspend control is specified, and channel-program suspension occurs when a suspend flag set to one is detected in a CCW.
When bit 4 is zero, suspend control is not specified, and the presence of a suspend flag set to one in any CCW of the channel program causes a program-check condition to be recognized.

**Streaming-Mode Control (C):** Bit 5 of word 1 controls streaming-mode enablement for subchannels configured to FICON-converted-I/O-interface channel paths during performance of the specified start function. When bit 5 is zero, the streaming mode is enabled at the subchannel. When bit 5 is one, the streaming mode is disabled at the subchannel. Bit 5 is meaningful only for subchannels configured to FICON-converted-I/O-interface channel paths and is ignored for subchannels configured to other channel-path types.

When the streaming mode is enabled, the channel path considers the first command of the designated channel program to be in progress at the associated device when the channel path receives the indication that the command has been accepted at the device. In addition, the channel path's acceptance of status, under certain conditions, is recognized by the channel path without receiving acknowledgement of status acceptance from the device.

When the streaming mode is not enabled, the channel path does not consider the first command of the designated channel program to be in progress at the associated device until the appropriate channel-path response, indicating that the device-command response or status has been accepted at the channel path, is sent to the device. In addition, when the device sends status to the channel path, the channel path's acceptance of that status is not recognized at the channel path until the channel path's confirmation of acceptance is received and acknowledged by the device.

**Modification Control (M):** Bit 6 of word 1 specifies whether modification control is required for the channel program. When bit 6 is zero, modification control is specified. When bit 6 is one, modification control is not specified.

When modification control is specified, the channel subsystem forces command synchronization with the addressed I/O device each time a command is executed and the previously executed command has the PCI and chain-command flags set to one and the chain-data and suspend flags set to zero.

When this condition is recognized, the channel subsystem signals a synchronization request to the I/O device for the current command. The channel subsystem temporarily suspends command chaining and does not fetch (or refetch) the next command-chained CCW until after normal ending status is received for the synchronizing command.

When modification control is not specified, then command synchronization is not required, and the channel subsystem may transfer commands to the I/O device without waiting for status.

The M bit is meaningful only for subchannels configured to FICON-I/O-interface or FICON-converted-I/O-interface channel paths and is ignored for other subchannels configured to other channel-path types.

**Programming Notes:**

1. FICON-converted-I/O-interface channel paths, modification control provides the capability to optimize dynamically modified channel programs that use the PCI flag in the CCW to initiate channel-program modification. Specifically, it allows the program to delay the channel-subsystem fetching and transferring of commands until after status is received for the command following the command with the PCI bit set. This increases the likelihood that a program-controlled interruption will be accepted by a CPU and acted upon by the program that dynamically modifies one or more command-chained CCWs that follow the synchronizing command.

In order to increase the probability that any dynamically modified CCWs are fetched after their modification and not prior to their modification, the modifying program should be executed as soon as possible following the CPU's acceptance of the program-controlled interruption. Additionally, the program should minimize the periods during which the configured CPUs are disabled for I/O interruptions.

For channel paths other than FICON-I/O-interface or FICON-converted-I/O-interface channel paths, command synchronization is implicit in the signalling protocol between the channel subsystem and the I/O device; therefore no explicit programming action is required to
force command synchronization. Regardless, the program should still attempt to accept and process program-controlled interruptions for these channel-path types in as timely a manner as possible for the same reason as stated above.

2. In order to allow the channel subsystem to optimize the execution of channel programs for FICON-I/O-interface or FICON-converted-I/O-interface channel paths, use of the modification-control facility is discouraged except for channel programs that require dynamic modification.

**Synchronization Control (Y):** Bit 7 of word 1 specifies whether synchronization control is required for the channel program. When bit 7 is zero and the prefetch-control bit, bit 9 of word 1, is one, synchronization control is specified. When bit 7 is one and bit 9 is one, synchronization control is not specified.

When synchronization control is specified, the channel subsystem forces command synchronization with the addressed I/O device whenever the current command in execution describes an input operation and the next CCW to be fetched describes an output operation. When this condition is recognized, the channel subsystem signals a synchronization request to the I/O device when the input command is transferred. The transfer of the output command is held pending at the subchannel until normal ending status, signaling the completion of the performance of the input operation by the I/O device, is received. Upon receipt of the ending status, the channel subsystem fetches (or refetches) the data associated with the output command and transfers it to the I/O device.

When synchronization control is not specified, the channel subsystem may transfer commands of the channel program without awaiting status that would signal the completion of the I/O operation for each command.

The Y bit is meaningful only when the subchannel is configured to FICON-I/O-interface or FICON-converted-I/O-interface channel paths and the prefetch-control bit, bit 9 of word 1, is one. The Y bit is ignored for subchannels configured to other channel-path types and when the prefetch-control bit is zero.

**Format Control (F):** Bit 8 of word 1 specifies the format of the channel-command words (CCWs) that make up the channel program designated by the channel-program-address field. When bit 8 of word 1 is zero, format-0 CCWs are specified. When bit 8 is one, format-1 CCWs are specified. (See “Channel-Command Word” on page 15-27 for the definition of the CCW formats.)

**Prefetch Control (P):** Bit 9 of word 1 specifies whether or not unlimited prefetching of CCWs is allowed for the channel program. When bit 9 is one, unlimited prefetching of CCWs is allowed. (Unlimited prefetching of data and IDAWs associated with the current and prefetched CCWs is always allowed.) It is model dependent whether prefetching is actually performed.

When bit 9 of word 0 is zero, no prefetching is allowed, except in the case of data chaining on output, where the prefetching of one CCW describing a data area is allowed. When bit 9 of word 0 is zero, the synchronization-control bit, bit 7 of word 1, is ignored.

Additional controls may limit the scope of prefetching.

**Initial-Status- Interruption Control (I):** Bit 10 of word 1 specifies whether or not the channel subsystem must verify to the program that the device has accepted the first command associated with a start or resume function. When the I bit is specified as one in the ORB, then, when the subchannel becomes active, indicating that the first command has been accepted for this start or resume function, the Z bit (see “Zero Condition Code (Z)” on page 16-11) is set to one at this subchannel, and the subchannel becomes status pending with intermediate status.

If the subchannel does not become active — for example, when the device signals channel end immediately upon receiving the first command, command chaining is not specified in the CCW, and command retry is not signaled — the command-accepted condition (Z bit set to one) is not generated; instead, the subchannel becomes status pending with primary status. Intermediate status may also be indicated in this case when the command is accepted if the first CCW contained the PCI flag set to one.
Address-Limit-Checking Control (A): When the address-limit-checking facility is installed, bit 11 of word 1 specifies whether or not address-limit checking is specified for the channel program. When this bit is zero, no address-limit checking is performed for the execution of the channel program, independent of the setting of the limit-mode bits in the subchannel (see "Limit Mode (LM)" on page 15-3). When this bit is one, address-limit checking is allowed for the channel program, subject to the setting of the limit-mode bits in the subchannel.

When the address-limit-checking facility is not installed, the address-limit-checking-control bit (A) must be zero in the ORB when START SUBCHANNEL is executed; otherwise, an operand exception is recognized.

Suppress-Suspended-Interruption Control (U): Bit 12 of word 1, when one, specifies that the channel subsystem is to suppress the generation of an intermediate interruption condition due to suspension if the subchannel becomes suspended. When bit 12 is zero, the channel subsystem generates an intermediate interruption condition whenever the subchannel becomes suspended during the execution of the channel program.

Format-2-IDAW Control (H): Bit 14 of word 1 specifies the format of IDAWs for CCWs that specify indirect data addressing. When bit 14 of word 1 is one, format-2 (64-bit data address) IDAWs are provided for all CCWs that have the IDAW flag set to one in the CCW. When bit 14 of word 1 is zero, format-1 (31-bit data address) IDAWs are provided for all CCWs that have the IDAW flag set to one in the CCW.

Programming Note: Sixty-four-bit IDAWs provide the only means by which data can be transferred directly between an I/O device and storage locations with addresses greater than 2G bytes.

2K-IDAW Control (T): Bit 15 of word 1 specifies the main-storage block size for format-2-IDAW data areas. Bit 15 is meaningful only when bit 14 (format-2 IDAW control) is one and is ignored when bit 14 is zero. When bit 15 of word 1 is one, all format-2 IDAWs designate 2K-byte storage blocks. When bit 15 of word 1 is zero, all format-2 IDAWs designate 4K-byte storage blocks.

Bit 13 of word 1 is reserved for future use and must be zero; otherwise, either an operand exception or a program-check condition is recognized.

Logical-Path Mask (LPM): Bits 16-23 of word 1 are preserved unmodified in the subchannel and specify to the channel subsystem which of the logical paths 0-7 are to be considered logically available, as viewed by the program. A bit setting of one means that the corresponding channel path is logically available; a zero specifies that the corresponding channel path is logically not available. If a channel path is specified by the program as being logically not available, the channel subsystem does not use that channel path to perform clear, halt, resume, or start functions when requested by the program, except when a dedicated-allegiance condition exists for that channel path. If a dedicated-allegiance condition exists, the setting of the LPM is ignored, and a resume, start, halt, or clear function is performed by using the channel path having the dedicated allegiance.

Incorrect-Length-Suppression Mode (L): When bit 8 of word 1 is one, then bit 24 of word 1, when one, specifies the incorrect-length-suppression mode. When the subchannel is in this mode when an immediate operation occurs (that is, when a device signals the channel-end condition during initiation of the command) and the current CCW contains a nonzero value in bit positions 16-31, indication of an incorrect-length condition is suppressed.

When bit 8 of word 1 is one, then bit 24 of word 1, when zero, specifies the incorrect-length-indication mode. When the subchannel is in this mode when an immediate operation occurs (that is, when a device signals the channel-end condition during initiation of the command) and the current CCW contains a nonzero value in bit positions 16-31, indication of an incorrect-length condition is recognized. Command chaining is suppressed unless the SLI flag in the CCW is one and the chain-data flag is zero.

When bit 8 of word 1 is zero, the value of bit 24 is ignored by the channel subsystem, and the subchannel is in the incorrect-length-suppression mode.
**ORB-Extension Control (X):** Bit 31 of word 1 specifies whether the ORB is extended. When bit 31 of word 1 is zero, the ORB consists of words 0-2, and words 3-7 are ignored. When bit 31 of word 1 is one, the ORB consists of words 0-7. Words 0 and 1 are described above. Words 2-7 are described below.

**Reserved:** Bits 25-30 of word 1 are reserved for future use and must be set to zeros. Bit 31 of word 1 must be zero if the ORB-extension facility is not installed. Otherwise, an operand exception or program-check condition is recognized.

**Channel-Program Address:** Bits 1-31 of word 2 specify the absolute address of the first CCW in main storage. Bit 0 of word 2 must be zero; otherwise, either an operand exception or a program-check condition is recognized. If format-0 CCWs are specified by bit 8 of word 1, then bits 1-7 of word 2 also must be zeros; otherwise, a program-check condition is recognized.

The three rightmost bits of the channel-program address must be zeros, designating the CCW on a doubleword boundary; otherwise, a program-check condition is recognized.

If the channel-program address designates a location protected against fetching or designates a location outside the storage of the particular installation, the start function is not initiated at the device. In this situation, the subchannel becomes status pending with primary, secondary, and alert status.

**Channel-Subsystem (CSS) Priority:** When bit 31 (X) of word 1 of the ORB is one, byte 0 of word 3 contains an unsigned binary integer, called the channel-subsystem-priority number, that is assigned to the designated subchannel and used to order the selection of subchannels when either a start function or a resume function is to be initiated for one or more subchannels that are start pending or resume pending.

The specified channel-subsystem-priority number can be any number in the range of 0 to 255. The numbers 0 and 255 designate the lowest and highest priorities, respectively.

Depending on the model and the configuration:

1. Fewer than 256 priority levels may be provided. For such models, the ORB-specified priority number may be ignored, and an alternative priority number may be implicitly assigned to the subchannel when the subchannel becomes start pending.
2. When bit 31 (X) of word 1 of the ORB is zero, an implicit priority number is assigned to the subchannel.

See “Channel-Subsystem-I/O-Priority Facility” on page 17-29 for details about how the priority number is assigned for both of these cases.

**Control-Unit (CU) Priority:** When bit 31 (X) of word 1 of the ORB is one, byte 2 of word 3 contains an unsigned binary integer, called the control-unit-priority number, that specifies, for an associated control unit attached by a FICON channel path, the priority level that is applied at the associated control unit for all I/O operations associated with the start function.

The specified control-unit-priority number can be any integer in the range of 1 to 255. The numbers 1 and 255 designate the lowest and highest priorities, respectively. The number 0 designates that no priority is assigned to the I/O operations associated with the start function. The handling of I/O operations when the priority number is 0 depends on the control-unit model.

Also depending on the control-unit model, fewer than 255 priority levels may be supported by the control unit. See the control-unit's System Library publication for additional information regarding the range of priority numbers supported and how this priority number is used.

The specified control-unit-priority number is ignored if any of the following conditions exists:

1. Bit 31 (X) of word 1 is zero. In this case, a control-unit-priority number of 0 is transmitted in the associated outbound frames.
2. The designated subchannel is not associated with a control unit configured to a FICON channel path.
3. The associated control unit does not provide prioritized performance of I/O operations. In this case, the control-unit-priority number in the associated outbound frames is ignored at the control unit.
4. The channel-subsystem model does not provide for the transmission of the control-unit-priority number.

5. The channel-subsystem-I/O-priority facility is not operational due to an operator action.

**Reserved:** All fields in the ORB that are defined as either “0” or “Reserved” must contain zeros when START SUBCHANNEL is executed; otherwise, either an operand exception or a program-check condition is recognized.

**Programming Notes:**

1. Bit positions of the ORB that presently are specified to contain zeros may in the future be assigned for the control of new functions.

2. The interruption parameter may contain any information, but ordinarily the information is of significance to the program handling the I/O interruption.

**Channel-Command Word**

The channel-command word (CCW) specifies the command to be executed and, for commands initiating certain I/O operations, it designates the storage area associated with the operation, the action to be taken whenever transfer to or from the area is completed, and other options.

A channel program consists of one or more CCWs that are logically linked such that they are fetched and executed by the channel subsystem in either a sequential or a nonsequential order. Sequential (contiguous) CCWs are linked by the use of the chain-data and chain-command flags, and nonsequential (noncontiguous) CCWs are linked by a CCW specifying the transfer-in-channel command.

As each CCW is executed, it is recognized as the current CCW. A CCW becomes current (1) when it is the first CCW of a channel program and has been fetched, (2) when, during command chaining, the new CCW is logically fetched, or (3) when, during data chaining, the new CCW takes over control of the I/O operation (see Data Chaining on page 15-33). When chaining is not specified, a CCW is no longer current after TEST SUBCHANNEL clears the start-function bit in the subchannel.

The location of the first CCW of the channel program is designated in the ORB that is the operand of START SUBCHANNEL. The first CCW is fetched subsequent to the execution of the instruction. The format of the CCWs fetched by the channel subsystem is specified by bit 8 of word 1 of the ORB. Each additional CCW in the channel program is obtained when the CCW is needed. Fetching of the CCWs by the channel subsystem does not affect those locations in main storage.

CCWs have either of two different formats, format 0 or format 1. The two formats do not differ in the information contained in the CCW, but they do differ in the size of the address and the arrangement of fields within the CCW.

The formats are defined as follows:

**Format-0 CCW**

<table>
<thead>
<tr>
<th>Cmd Code</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>39 48 63</td>
</tr>
</tbody>
</table>

**Format-1 CCW**

<table>
<thead>
<tr>
<th>Cmd Code</th>
<th>Flags</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Count</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 34 36 38 (in format-0 CCW)</td>
</tr>
<tr>
<td>8 10 12 14 (in format-1 CCW)</td>
</tr>
</tbody>
</table>

Format-0 CCWs can be located anywhere in the first $2^{24}$ (16M) bytes of absolute storage, and format-1 CCWs can be located anywhere in the first $2^{31}$ (2G) bytes of absolute storage.

Bit 39 (format 0) or bit 15 (format 1) of every CCW other than a format-0 CCW specifying transfer in channel must be zero. If indirect data addressing is specified and the format-2-IDAW-control bit is zero in the ORB associated with the CCW, then:
1. Bits 30 and 31 (format 0) or 62 and 63 (format 1) of the CCW must be zeros, designating a word boundary.
2. Bit 0 of the first entry of the indirect-data-address list must be zero.

If indirect data addressing is specified and the format-2-IDAW-control bit is one in the ORB associated with the CCW, bits 29-31 (format 0) or bits 61-63 (format 1) of the CCW must be zeros, designating a doubleword boundary. When any of these requirements is not met, a program-check condition may be recognized (see “CCW Indirect Data Addressing” on page 15-36). Detection of this condition during data chaining causes the I/O device to be signaled to conclude the operation. When the absence of these zeros is detected during command chaining or subsequent to the execution of START SUBCHANNEL, the new operation is not initiated, and an interruption condition is generated.

The contents of bit positions 40-47 of a format-0 CCW are ignored.

The fields in the CCWs are defined as follows:

**Command Code:** Bits 0-7 (both formats) specify the operation to be performed.

**Data Address:** Bits 8-31 (format 0) or bits 33-63 (format 1) designate a location in absolute storage. The designated location is the first location referred to in the area designated by the CCW. Bit 32 of a format-1 CCW must be zero; otherwise, a program-check condition is recognized. If a byte count of zero is specified, this field is not checked.

See the section “CCW Indirect Data Addressing” on page 15-36 for information about the specification of data addresses greater than 2G bytes.

**Chain-Data (CD) Flag:** Bit 32 (format 0) or bit 8 (format 1), when one, specifies chaining of data. The bit causes the storage area designated by the next CCW to be used with the current I/O operation. When the CD flag is one in a CCW, the chain-command and suppress-length-indication flags (see below) are ignored.

**Chain-Command (CC) Flag:** Bit 33 (format 0) or bit 9 (format 1), when one, and when the CD flag and S flag are both zeros, specifies chaining of commands. The bit causes the operation specified by the command code in the next CCW to be initiated upon the normal completion of the current operation.

**Suppress-Length-Indication (SLI) Flag:** Bit 34 (format 0) or bit 10 (format 1) controls whether an incorrect-length condition is to be indicated to the program. When this bit is one and the CD flag is zero, the incorrect-length indication is suppressed. When both the CC and SLI flags are ones and the CD flag is zero, command chaining takes place, regardless of the presence of an incorrect-length condition. This bit should be specified in all CCWs where suppression of the incorrect-length indication is desired.

**Skip (SKP) Flag:** Bit 35 (format 0) or bit 11 (format 1), when one, specifies the suppression of transfer of information to storage during a read, read-backward, sense ID, or sense operation.

**Program-Controlled-Interruption (PCI) Flag:** Bit 36 (format 0) or bit 12 (format 1), when one, causes the channel subsystem to generate an intermediate interruption condition when the CCW containing the bit takes control of the I/O operation. When the PCI flag bit is zero, normal operation takes place.

**Indirect-Data-Address (IDA) Flag:** Bit 37 (format 0) or bit 13 (format 1), when one, specifies indirect data addressing.

**Suspend (S) Flag:** Bit 38 (format 0) or bit 14 (format 1), when one, specifies suspension of channel-program execution. When valid, it causes channel-program execution to be suspended prior to the execution of the CCW containing the S flag. A one value of the S flag is valid when bit 4 of word 1 of the associated ORB is one.

**Count:** Bits 48-63 (format 0) or bits 16-31 (format 1) specify the number of bytes in the storage area designated by the CCW.

**Programming Note:** Bit 39 of a format-0 CCW or bit 15 of a format-1 CCW, which presently must be zero, may in the future be assigned for the control of new functions. It is recommended, therefore, that this bit not be set to one for the purpose of obtaining an intentional program-check indication.
Command Code

The command code, bit positions 0-7 of the CCW, specifies to the channel subsystem and the I/O device the operation to be performed.

The two rightmost bits or, when these bits are zeros, the four rightmost bits of the command code identify the operation to the channel subsystem. The channel subsystem distinguishes among the following four operations:

- Output forward (write, control)
- Input forward (read, sense, sense ID)
- Input backward (read backward)
- Branching (transfer in channel)

The channel subsystem ignores the leftmost bits of the command code, except in a format-1 CCW specifying transfer in channel. In this case, all bits of the command code are decoded by the channel subsystem.

Commands that initiate I/O operations (write, read, read backward, control, sense, and sense ID) cause all eight bits of the command code to be transferred to the control unit. In these command codes, the leftmost bit positions contain modifier bits. The modifier bits specify to the device how the command is to be executed. They may, for example, cause the device to compare data received during a write operation with data previously recorded, and they may specify such attributes as recording density and parity. For the control command, the modifier bits may contain the order code specifying the control function to be performed. The meaning of the modifier bits depends on the type of I/O device and is specified in the System Library publication for the device.

The command-code assignment is listed in Figure 15-5. The symbol x indicates that the bit position is ignored; m identifies a modifier bit.

<table>
<thead>
<tr>
<th>Code</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>x x x x 0 0 0 0</td>
<td>Invalid</td>
</tr>
<tr>
<td>m m m m 0 0 1 0</td>
<td>Write</td>
</tr>
<tr>
<td>m m m m 0 0 1 0</td>
<td>Read</td>
</tr>
<tr>
<td>m m m m 1 0 0 0</td>
<td>Read backward</td>
</tr>
<tr>
<td>m m m m m 1 1 1</td>
<td>Control</td>
</tr>
<tr>
<td>m m m m 0 1 0 0</td>
<td>Sense</td>
</tr>
<tr>
<td>1 1 1 0 0 1 0 0</td>
<td>Sense ID</td>
</tr>
<tr>
<td>x x x x 1 0 0 0</td>
<td>Transfer in channel¹</td>
</tr>
<tr>
<td>0 0 0 0 1 0 0 0</td>
<td>Transfer in channel²</td>
</tr>
<tr>
<td>m m m m 1 0 0 0</td>
<td>Invalid³</td>
</tr>
</tbody>
</table>

**Explanation:**

- m Modifier bit
- x Ignored
- ¹ Format-0 CCW
- ² Format-1 CCW
- ³ Format-1 CCW with any of bits 0-3 nonzero

Figure 15-5. Command-Code Assignment

Whenever the channel subsystem detects an invalid command code during the initiation of command execution, the program-check-interruption condition is generated, and channel-program execution is terminated. The command code is ignored during data chaining, unless it specifies transfer in channel.

Designation of Storage Area

The main-storage area associated with an I/O operation is defined by one or more CCWs. A CCW defines an area by specifying the address of the first byte to be transferred and the number of consecutive bytes contained in the area. The address of the first byte of data to be transferred is specified either directly in the data-address field of the CCW or indirectly in an indirect-data-address word (IDAW) designated by the data-address field of the CCW. The number of bytes contained in the storage area is specified in the count field.

In write, read, control, and sense operations, storage locations are used in ascending order of addresses. As information is transferred to or from main storage, the address from the address field is incremented, and the count from the count field is decremented. The read-backward operation places data in storage in a descending order of addresses, and both the count and the address
are decremented. When the count reaches zero, the storage area defined by the CCW is exhausted.

Any main-storage location available to the start function can be used in the transfer of data to or from an I/O device, provided that the location is not protected against that type of reference. Format-0 CCWs can be located in any available part of the first $2^{14} (16M)$ bytes of absolute storage, and format-1 CCWs can be located in any available part of the first $2^{31} (2G)$ bytes of absolute storage, provided that, in both cases, the location is not protected against a fetch-type reference. When the channel subsystem attempts to refer to a protected location, the protection-check condition is generated, and the device is signaled to terminate the operation.

A main-storage location is available if it is available in the configuration and, in the case of a data location (not a CCW or IDAW), access to it is not prevented by the address-limit-checking facility. If a main-storage location is not available, it is said to have an invalid address.

If the channel subsystem refers to a location that is not available, the program-check condition is generated. When the first CCW designated by the channel-program address is at an unavailable location, the start function is not initiated at the device, the status portion of the SCSW is updated with the program-check indication, the subchannel becomes status pending with primary, secondary, and alert status, and deferred condition code 1 is indicated. Invalid data addresses, as well as any invalid CCW addresses detected on chaining or subsequent to the execution of START SUB-CHANNEL, cause the channel subsystem to signal the device to stop, and the I/O operation is terminated. The channel subsystem sets the incorrect-length condition if the SLI flag is not one in the CCW. No data is transferred. If the device does not request data transfer, the operation proceeds to the normal ending point.

If a zero byte count is contained in a format-0 CCW that does not specify transfer in channel, or if a zero byte count is contained in a format-1 CCW that specifies data chaining or was fetched while data chaining, a program-check condition is recognized, and the subchannel is made status pending with combinations of primary, secondary, and alert status as a function of the state of the subchannel and the status received from the device.

During an output operation, the channel subsystem may fetch one CCW describing a data area at any time during the execution of the current CCW. If unlimited prefetching is allowed by the setting of the prefetch-control bit in the ORB, any number of CCWs and IDAWs and the associated data may be prefetched by the channel subsystem. When the I/O operation uses data and CCWs from locations near the end of the available storage, such prefetching may cause the channel subsystem to refer to locations that do not exist. Invalid addresses detected during prefetching do not affect the performance of the I/O operation and do not cause error indications until the operation actually attempts to use the information. If the operation is concluded by the I/O device or by the execution of HALT SUB-CHANNEL or CLEAR SUBCHANNEL before the invalid information is needed, the condition is not brought to the attention of the program.

The count field in the CCW can specify any number of bytes up to 65,535. In format-0 CCWs, the count field is always nonzero unless the command code specifies transfer in channel, in which case the count field is ignored. In format-1 CCWs, the count field may contain the value zero unless data chaining is specified or the CCW is fetched while data chaining. Whenever (1) the count field in a format-1 CCW is zero, (2) data chaining is either not specified or not in effect, and (3) data transfer is requested by the device, the device is signaled to stop, and the I/O operation is terminated. The channel subsystem sets the incorrect-length condition if the SLI flag is not one in the CCW. No data is transferred. If the device does not request data transfer, the operation proceeds to the normal ending point.

If a zero byte count is contained in a format-0 CCW that does not specify transfer in channel, or if a zero byte count is contained in a format-1 CCW that specifies data chaining or was fetched while data chaining, a program-check condition is recognized, and the subchannel is made status pending with combinations of primary, secondary, and alert status as a function of the state of the subchannel and the status received from the device.

Note: For a description of the storage area associated with a CCW when indirect data addressing is used, see "CCW Indirect Data Addressing" on page 15-36.
Programming Notes:

1. Since a format-1 CCW with a count of zero is valid, the program can use the CCW count field to specify that no data be transferred to the I/O device. If the device requests a data transfer, the device is signaled to terminate data transfer. If the SLI and chain-command flags are also specified as ones, and no unusual conditions are encountered subsequent to signaling the device to terminate data transfer, the new operation is initiated upon receipt of device end from the device.

2. If the subchannel is in the incorrect-length-suppression mode, the chain-data flag in the current CCW is zero, and the operation is performed as an immediate operation, then incorrect length is not indicated, regardless of the setting of the SLI flag.

If the subchannel is in the incorrect-length-indication mode, if the chain-data flag in the current CCW is zero, and if the operation is performed as an immediate operation, then incorrect length is indicated if the count field of the current CCW specifies a nonzero value, unless suppressed by the SLI flag of the CCW; incorrect length is not indicated, however, if the count field of the CCW specifies a value of zero.

If a new CCW that has a count field of zero is fetched during data chaining or if a CCW is fetched with the chain-data flag set to one and a count field of zero, a program-check condition is recognized by the channel subsystem.

Chaining

When the channel subsystem has completed the transfer of information specified by a CCW, it can continue performing the start function by fetching a new CCW. Such fetching of a new CCW is called chaining, and the CCWs belonging to such a sequence are said to be chained.

Chaining takes place between CCWs located in successive doubleword locations in storage. It proceeds in an ascending order of addresses; that is, the address of the new CCW is obtained by adding 8 to the address of the current CCW. Two chains of CCWs located in noncontiguous storage areas can be coupled for chaining purposes by a transfer-in-channel command. All CCWs in a chain apply to the I/O device that is associated with the subchannel designated by the original START SUBCHANNEL instruction.

Two types of chaining are provided: chaining of data and chaining of commands. Chaining is controlled by the chain-data (CD) and chain-command (CC) flags in conjunction with the suppress-length-indication (SLI) flag in the CCW. These flags specify the action to be taken by the channel subsystem upon the exhaustion of the current CCW and upon receipt of ending status from the device, as shown in Figure 15-6 on page 15-32.

The specification of chaining is effectively propagated through a transfer-in-channel command. When, in the process of chaining, a transfer-in-channel command is fetched, the CCW designated by the transfer-in-channel command is used for the type of chaining specified in the CCW preceding the transfer-in-channel command.

The CD and CC flags are ignored in a format-0 CCW specifying the transfer-in-channel command. In a format-1 CCW specifying the transfer-in-channel command, the CD and CC flags must be zeros; otherwise, a program-check condition is recognized.
### Table: Action at the Subchannel upon Exhaustion of Count or Receipt of Channel End

<table>
<thead>
<tr>
<th>Flags in Current CCW</th>
<th>Action at the Subchannel upon Exhaustion of Count or Receipt of Channel End</th>
<th>Immediate Operation</th>
<th>Non-immediate Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incorrect-Length-Suppression Mode&lt;sup&gt;1&lt;/sup&gt;</td>
<td>Incorrect-Length- Indication Mode</td>
<td>Count Exhausted</td>
</tr>
<tr>
<td>CD</td>
<td>CC</td>
<td>SLI</td>
<td>CCW Count=0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>End, NIL</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>End, NIL</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CC</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>CC</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>End, NIL</td>
</tr>
</tbody>
</table>

### Explanation:

- The selected bit is ignored and may be either zero or one.

* These situations cannot validly occur. When data chaining is specified, the new CCW takes control of the operation after transferring the last byte of data designated by the current CCW, but before the next request for data or status transfer from the device. The new CCW (which cannot contain a count of zero unless a program-check condition is also recognized) is in control of the operation.

<sup>1</sup> The count field must contain a nonzero value when format-0 CCWs are specified; otherwise, the operation is terminated with a program-check condition.

CD Command chaining is performed by the channel subsystem upon receipt of device end.

CD The chain-data flag causes the channel subsystem to immediately fetch a new CCW for the same operation. The operation continues unless the CCW thus fetched has a count field of zero, in which case the operation is terminated with a program-check condition.

CE Channel end from the device that indicates end of block.

End Operation is terminated.

IL Incorrect length is indicated with the subsequent interruption condition generated at the subchannel.

NIL Incorrect length is not indicated with the subsequent interruption condition generated at the subchannel.

PC These situations cannot validly occur. The channel subsystem recognizes a program-check condition when a CCW is fetched that has the chain-data flag set to one and a count field of zero.

Stop Device is signaled to terminate data transfer, but subchannel remains subchannel active until channel end is received.

---

*Figure 15-6. Subchannel Chaining Action*
Programming Note: When bit 9 of word 1 of the ORB is one, unlimited prefetching of chained CCWs (including CCWs linked by a transfer-in-channel command) by the channel subsystem is permitted. When prefetching is allowed by the ORB, no modification of the channel program should be performed after START SUBCHANNEL is executed and before the primary interruption condition for the operation has been received unless the subchannel is currently suspended and is not resume pending.

Data Chaining
During data chaining, the new CCW fetched by the channel subsystem defines a new storage area for the original I/O operation. If the channel path is of the parallel-I/O-interface type, the performance of the operation at the I/O device is not affected. If the channel path is of the serial-I/O-interface type, then the performance of the operation at the I/O device either is not affected or, depending on the device model, may be terminated with unit-check status. When the operation at the I/O device is not affected and all data designated by the current CCW has been transferred to main storage or to the device, data chaining causes the operation to continue, using the storage area designated by the new CCW. The contents of the command-code field of the new CCW are ignored, unless they specify transfer in channel.

Data chaining is considered to occur immediately after the last byte of data designated by the current CCW has been transferred to main storage or to the device. When the last byte of the data transfer has been placed in main storage or accepted by the device, the new CCW takes over the control of the operation. If the device sends channel end after exhausting the count of the current CCW but before transferring any data to or from the storage area designated by the new CCW, the SCSW associated with the concluded operation pertains to the new CCW.

If programming errors are detected in the new CCW or during its fetching, the error indication is generated, and the device is signaled to conclude the operation when it attempts to transfer data designated by the new CCW. If the device signals the channel-end condition before transferring any data designated by the new CCW, program check or protection check is indicated in the SCSW associated with the termination. The contents of the SCSW pertain to the new CCW unless the address of the new CCW is invalid, the location is protected against fetching, or programming errors are detected in an intervening transfer-in-channel command. A data address referring to a non-existent or protected area causes an error indication only after the I/O device has attempted to transfer data to or from the invalid location.

Data chaining during an input operation causes the new CCW to be fetched when all data designated by the current CCW has been placed in main storage. On an output operation, the channel subsystem may fetch the new CCW from main storage before data chaining occurs. Any programming errors in the prefetched CCW, however, do not affect the performance of the operation until all data designated by the current CCW has been transferred to the I/O device. If the device concludes the operation before all data designated by the current CCW has been transferred, the conditions associated with the prefetched CCW are not indicated to the program. Unlimited prefetching is allowed under the control of the prefetch bit specified in the ORB. (See "Prefetch Control (P)" on page 15-24.) When unlimited prefetching is not allowed and an output operation is specified, only one CCW describing a data area may be prefetched. If a prefetched CCW specifies transfer in channel, only one more CCW may be fetched before the exhaustion of the current CCW.

Programming Notes:

1. If the ORB does not specify unlimited prefetching, no prefetching of CCWs is performed, except in the case of data chaining on an output operation where one CCW describing a data area may be prefetched at a time.

If the ORB for the I/O operation specifies that prefetching is allowed, any number of CCWs and IDAWs and the associated data areas may be prefetched and buffered in the channel subsystem.

The same actions for signaling errors and terminating operations take place when unlimited prefetching is allowed by the ORB as when it is not allowed. However, when unlimited prefetching is specified and an error condition is detected, both the channel subsystem and the program must recognize that the points of ter-
mination at the channel subsystem and at the I/O device may be different in terms of the channel command in execution at the point of error. The channel subsystem indicates the point of termination at the channel subsystem by storing the appropriate CCW address in word 1 of the subchannel-status word and the point of termination at the device by storing the secondary-CCW address in word 4 of the format-0 extended-status word.

When prefetching has been specified in the ORB, the result of modifications to CCWs after START SUBCHANNEL has been executed or after self-describing channel programs have been used is unpredictable. (See note 2 for the definition of self-describing channel programs.)

2. Data chaining may be used to rearrange information as it is transferred between main storage and an I/O device. Data chaining permits blocks of information to be transferred to or from noncontiguous areas of storage, and, when used in conjunction with the skipping function, data chaining allows the program to place in main storage specified portions of a block of data.

When, during an input operation, the program specifies data chaining to a location in which data has been placed under the control of the current CCW, the channel subsystem, in fetching the next CCW, fetches the new contents of the location. This is true even if the location contains the last byte transferred under the control of the current CCW. When a channel program data-chains to a CCW placed in storage by the CCW specifying data chaining, the input block is said to be self-describing. A self-describing block contains one or more CCWs that designate storage locations and counts for subsequent data in the same input block.

The use of self-describing blocks is equivalent to the use of unchecked data. An I/O data-transfer malfunction that affects validity of a block of information is signaled only at the completion of data transfer. The error condition normally does not prematurely terminate or otherwise affect the performance of the operation. Thus, there is no assurance that a CCW read as data is valid until the operation is completed. If the CCW thus read is in error, use of the CCW in the current operation may cause subsequent data to be placed at wrong locations in main storage with resultant destruction of its contents, subject only to the control of the protection key and the address-limit-checking facility, if used.

3. When, during data chaining, a device transfers data by using the data-streaming feature, an overrun or chaining-check condition may be recognized when a small byte-count value is specified in the CCW. The minimum acceptable number of bytes that can be specified varies as a function of the system model and system activity.

Command Chaining

During command chaining, the new CCW fetched by the channel subsystem specifies a new I/O operation. The channel subsystem fetches the new CCW upon the receipt of the device-end signal for the current operation. If the new CCW does not have its S flag set to one and no unusual conditions are detected, the channel subsystem initiates the new operation. The presence of the S flag set to one or unusual conditions causes command chaining to be suppressed. When command chaining takes place, the completion of the current operation does not cause an I/O interruption, and the count indicating the amount of data transferred during the current operation is not made available to the program. For operations involving data transfer, the new command always applies to the next block of data at the device.

Command chaining takes place and the new operation is initiated only if no unusual conditions have been detected in the current operation. In particular, the channel subsystem initiates a new I/O operation by command chaining upon receipt of a status byte containing only the following bit combinations: (1) device end, (2) device end and status modifier, (3) device end and channel end, and (4) device end, channel end, and status modifier. In the first two cases, channel end is signaled before device end, with all other status bits zeros. If a condition such as attention, unit check, unit exception, incorrect length, program check, or protection check has occurred, the sequence of operations is concluded, and the status associated with the current operation causes an interruption condition to be generated. The new CCW in this case is not fetched. The incorrect-length condition does not suppress command chaining if the current CCW has the SLI flag set to one.
An exception to sequential chaining of CCWs occurs when the I/O device presents the status-modifier condition with the device-end signal or channel-end and device-end signals. When command chaining is specified and no unusual conditions have been detected, or when command retry has been previously signaled and an immediate retry could not be performed, the combination of status-modifier and device-end bits causes the channel subsystem to alter the sequential execution of CCWs. If command chaining was specified, status modifier and device end cause the channel subsystem to fetch and chain to the CCW whose main-storage address is 16 higher than that of the CCW that specified chaining. If command retry was previously signaled and immediate retry could not be performed, the status causes the channel subsystem to command chain to the CCW whose storage address is 8 higher than that of the CCW for which retry was initially signaled.

When both command and data chaining are specified, the first CCW associated with the operation specifies the operation to be performed, and the last CCW specifies whether another operation follows.

**Programming Note:** Command chaining makes it possible for the program to initiate transfer of multiple blocks of data by issuing a single START SUBCHANNEL instruction. It also permits a subchannel to be set for execution of other commands, such as positioning the disk-access mechanism, and for data-transfer operations without interference by the program at the end of each operation. Command chaining, in conjunction with the status-modifier condition, permits the channel subsystem to modify the normal sequence of operations in response to signals provided by the I/O device.

**Skipping**

Skipping causes the suppression of main-storage references during an I/O operation. It is defined only for read, read-backward, sense-ID, and sense operations, and is controlled by the skip flag, which can be specified individually for each CCW. When the skip flag is one, skipping occurs; when it is zero, normal operation takes place. The setting of the skip flag is ignored in all other operations. Skipping affects only the handling of information by the channel subsystem. The operation at the I/O device proceeds normally, and information is transferred. The channel subsystem keeps updating the count but does not place the information in main storage. Chaining is not precluded by skipping. In the case of data chaining, normal operation is resumed if the skip flag in the new CCW is zero.

No checking for invalid or protected data addresses takes place during skipping.

**Programming Note:** Skipping, when combined with data chaining, permits the program to place in main storage specified portions of a block of information from an I/O device.

**Program-Controlled Interruption**

The program-controlled-interruption (PCI) function permits the program to cause an I/O interruption during the performance of an I/O operation. The function is controlled by the PCI flag of the CCW. Neither the value of the PCI flag nor the associated interruption request affects the performance of the current operation.

The value of the PCI flag can be one either in the first CCW designated for the current start or resume function or in a CCW fetched during chaining. If the PCI flag is one in a CCW that has become current, the subchannel becomes status pending with intermediate status, and an I/O-interruption request is generated. The point at which the subchannel becomes status pending depends on the progress of the current start or resume function as follows:

1. If the PCI flag is one in the first CCW associated with a start function or a resume function, the subchannel becomes status pending with intermediate status only after the command has been accepted.
2. If the PCI flag is one in a CCW that has become current while data chaining, the subchannel becomes status pending with intermediate status after all data designated by the preceding CCW has been transferred.
3. If the PCI flag is one in a CCW that has become current while command chaining, the subchannel becomes status pending with intermediate status as that CCW becomes current.
In all cases, if the subchannel is enabled for I/O interruptions, the point of interruption depends on the current activity in the system and may be delayed. No predictable relationship exists between the point at which the interruption request is generated because of the PCI flag and the extent to which data transfer has been completed to or from the area designated by the CCW. However, all the fields within the SCSW pertain to the same instant.

An intermediate interruption condition that is made pending because of a PCI flag remains pending during chaining if not cleared by TEST SUBCHANNEL or CLEAR SUBCHANNEL. If another CCW containing a PCI flag that is one becomes current prior to the clearing of the intermediate interruption condition, only one interruption condition is preserved.

An intermediate interruption may occur while the subchannel is subchannel-and-device active with the operation specified by the CCW causing the intermediate interruption condition or with the operation specified by a CCW that has subsequently become current. If the intermediate interruption condition is not cleared prior to the conclusion of the operation or chain of operations, the condition is indicated together with the primary interruption condition at the conclusion of the operation or chain of operations. The intermediate interruption condition may be cleared by TEST SUBCHANNEL while the subchannel is subchannel active.

If the SCSW stored by TEST SUBCHANNEL indicates that the subchannel is status pending with intermediate status and the operation or chain of operations has not been concluded (that is, the activity-control field indicates subchannel-and-device active or suspended), then the CCW-address field contains an address that is 8 higher than the address of the most recent CCW to become current and have a PCI flag that is one, or the CCW-address field contains an address that is 8 higher than the address of a CCW that has subsequently become current. Unless the SCSW also contains the primary-status bit set to one, the device-status field contains zeros, and the count is unpredictable.

Subchannel-status conditions other than PCI may be indicated when the SCSW is stored. If the subchannel is not also status pending with primary status, these conditions may or may not be indicated again. If the subchannel-status condition is detected while prefetching and the operation or chain of operations is concluded before the condition affects an operation, the condition is reset and is not indicated when the subchannel subsequently becomes status pending with primary status. If the subchannel-status condition affects an operation, the condition is indicated when the subchannel becomes status pending with primary status.

If the program-controlled-interruption condition remains pending until the operation or chain of operations is concluded at the subchannel, a single interruption request exists. When TEST SUBCHANNEL is subsequently executed, the status-control field of the SCSW stored indicates both the primary interruption condition and the intermediate interruption condition, and the PCI bit of the subchannel-status field is one.

The value of the PCI flag is inspected in every CCW except for those CCWs that specify the transfer-in-channel command. The PCI flag is ignored during initial program loading.

**Programming Notes:**

1. The program-controlled interruption provides a means of alerting the program to the progress of chaining during an I/O operation. It permits programmed dynamic main-storage allocation.

2. A CCW with a PCI flag set to one may, if retried because of command retry, cause multiple PCI interruptions to occur. (See "Command Retry" on page 15-42)

**CCW Indirect Data Addressing**

CCW indirect data addressing permits a single channel-command word to control the transfer of data that spans noncontiguous 2K-byte or 4K-byte blocks in main storage. The use of CCW indirect data addressing also allows the program to designate data addresses above 16M bytes when using format-0 CCWs or above 2G bytes when using format-1 CCWs.

CCW indirect data addressing is specified by a flag in the CCW which, when one, indicates that the data address is not used to directly address data. Instead, the address points to a list of words or doublewords, called indirect-
data-address words (IDAWs), each of which contains an absolute address designating a data area in main storage.

IDAWs have either of two formats, called format 1 and format 2, as determined by the format-2-IDAW control, bit 14 of word 1 of the ORB associated with the channel program being executed. When the format-2-IDAW control is zero, the IDAW is format 1 and is a word containing a 31-bit address. When the control is one, the IDAW is format 2 and is a doubleword containing a 64-bit address. The IDAW formats are as follows:

**Format-1 IDAW**

```
    │ Data Address  
0 31
```

**Format-2 IDAW**

```
    │ Data Address (Bytes 0-3)  
0 31

    │ Data Address (Bytes 4-7)  
32 63
```

Bit 0 (format 1) is reserved for future use and must be zero; otherwise, a program-check condition may be recognized, as described below.

A format-1 IDAW designates a data area within a 2K-byte block of main storage and is capable of addressing storage in the range of 0 to \(2^{11} - 1\).

A format-2 IDAW designates a data area within a 2K-byte or 4K-byte block of main storage, as determined by the 2K-IDAW control, bit 15 of word 1 of the ORB associated with the channel program being executed, and is capable of addressing storage in the range of 0 to \(2^{64} - 1\). When the 2K-IDAW-control bit is zero, each format-2 IDAW of the designated channel program designates a 4K-byte block of main storage. When the 2K-IDAW-control bit is one, each format-2 IDAW designates a 2K-byte data-area block. All IDAWs associated with the designated channel program must have the same IDAW format, and all of those IDAWs specify the same size of storage block.

When the indirect-data-addressing bit in the CCW is one, the data-address field of the CCW designates the location of the first IDAW to be used for data transfer for the command. Additional IDAWs, if needed for completing the data transfer for the CCW, are in successive locations in storage. The number of IDAWs required for a CCW is determined by the IDAW format as specified in the ORB, by the count field of the CCW, and by the data address in the initial IDAW. When, for example, (1) the ORB specifies format-2 IDAWs with 4K-byte blocks, (2) the CCW count field specifies 8K bytes, and (3) the first IDAW designates a location in the middle of a 4K-byte block, then three IDAWs are required.

The IDAW designated by the CCW can designate any location. Data is then transferred, for read, write, control, sense ID, and sense commands, to or from successively higher storage locations or, for a read-backward command, to successively lower storage locations, until a 2K-byte block boundary (format-1 or format-2 IDAW) or a 4K-byte block boundary (format-2 IDAW) is reached. The control of data transfer is then passed to the next IDAW. The second and any subsequent IDAWs must designate, depending on the command, the first byte, or the last byte for read backward, of a 2K-byte block (format-1 or format-2 IDAW) or a 4K-byte block (format-2 IDAW). Thus, for read, write, control, sense ID, and sense commands, such format-1 IDAWs must have zeros in bit positions 21-31, and such format-2 IDAWs must have zeros in bit positions 53-63 (2K-byte blocks) or 52-63 (4K-byte blocks). For a read-backward command, such format-1 IDAWs must have ones in bit positions 21-31, and such format-2 IDAWs must have ones in bit positions 53-63 (2K-byte blocks) or 52-63 (4K-byte blocks). If any of these rules is violated, a program-check condition is recognized.

Except for the unique restrictions on the designation of the data address by the IDAW, all other actions taken for the data address, such as for protected storage and invalid addresses, and the actions taken for data prefetching are the same as when indirect data addressing is not used.

IDAWs pertaining to the current CCW or a prefetched CCW may be prefetched. The number of IDAWs that can be prefetched cannot exceed that required to satisfy the count in the CCW that points to the IDAWs. An IDAW takes control of...
data transfer when the last byte has been transferred for the previous IDAW. The same actions take place as with data chaining regarding when an IDAW takes control of data transfer during an I/O operation. That is, when the count for the CCW has not reached zero, a new IDAW takes control of the data transfer when the last byte has been transferred for the previous IDAW for that CCW, even in situations where (1) channel end, (2) channel end and device end, or (3) channel end, device end, and status modifier are received prior to the transfer of any data bytes pertaining to the new IDAW.

A prefetched IDAW does not take control of an I/O operation if the count in the CCW has reached zero with the transfer of the last byte of data for the previous IDAW for that CCW. Program or access errors detected in prefetched IDAWs are not indicated to the program until the IDAW takes control of data transfer. However, when the channel subsystem detects an invalid CBC on the contents of a prefetched IDAW or its associated key, the condition may be indicated to the program, when detected, before the IDAW takes control of data transfer. For a description of the indications provided when an invalid CBC is detected on the contents of an IDAW or its associated key, see "Channel-Control Check" on page 16-27.

Bits 1-31 (format 1) or bits 0-63 (format 2) designate the absolute storage location of the first byte to be used in the data transfer. When format-1 IDAWs are specified, the channel subsystem forms a 64-bit absolute main-storage address by appending 33 zero bits on the left of bit 1.

When the IDAW flag of the CCW is set to one and any of the following conditions occurs:

1. Format-1 IDAWs are specified in the ORB, and the address in the CCW does not designate the first IDAW on a word boundary,
2. Format-2 IDAWs are specified in the ORB, and the address in the CCW does not designate the first IDAW on a doubleword boundary,
3. The address in the CCW designates a storage location that is not physically available,
4. Access to the storage location specified by the address in the CCW is prohibited by protection, or
5. Bit 0 (format 1 only) of the first IDAW is not zero,
then, depending on the model, one of the following two actions is taken independent of the setting of the skip flag (if condition 5 above is true, action 2 must be taken).

1. The above conditions are checked before initiating the operation at the device. If any of these conditions is recognized, initiation of the I/O operation does not occur, and the subchannel is made status pending with primary, secondary, and alert status.

2. The operation is initiated at the device prior to checking for these conditions. If the device attempts to transfer data, the device is signaled to terminate the I/O operation, and the subchannel is made status pending with primary, secondary, and alert status as a function of the subchannel state and the status presented by the device.

Suspension of Channel-Program Execution

The suspend function, when used in conjunction with RESUME SUBCHANNEL, provides the program with a means to stop and restart the execution of a channel program. The initiation of the suspend function is controlled by the setting of the suspend control, bit 4 of word 1 of the ORB. The suspend function is signaled when suspend control has been specified for the subchannel in the ORB and a CCW containing an S flag set to one becomes the current CCW. The flag can be indicated either in the first CCW of the channel program or in a CCW fetched while command chaining. The S flag is not valid and causes a program-check condition to be recognized if (1) the ORB contains the suspend-control bit set to zero, or (2) the CCW is fetched while data chaining (see "Data Chaining" on page 15-33, concerning the handling of programming errors detected during data chaining).

Upon recognition of the suspend function, suspension of channel-program execution occurs when the CCW becomes current (see "Channel-Command Word" on page 15-27 for a definition of when a CCW becomes current). If suspension occurs during command chaining, the device is signaled that command chaining is no longer in effect.
RESUME SUBCHANNEL signals that the CCW that caused channel-program suspension may have been modified, that the CCW must be refetched, and that the contents of the CCW must be examined to determine the settings of the flags. If the S flag is one, execution of that CCW does not occur. If the CCW is valid and the S flag in the CCW is zero, execution is initiated (see “RESUME SUBCHANNEL” on page 14-10 and “Start Function and Resume Function” on page 15-18). When a valid CCW that contains an S flag validly set to one becomes the current CCW during command chaining and the resume-pending condition is not recognized, the suspend function is performed and causes the following actions to occur in the order given:

1. The device is signaled that the chain of operations has been concluded.
2. Channel-program execution is suspended at the subchannel; all prefetched IDAWs, CCWs, and data are discarded; and the subchannel is set up such that the resume function can be performed when the subchannel is next recognized to be resume pending.
3. If the measurement-block-update mode is active and the subchannel is enabled for the mode, the accrued values of the measurement data, including the start-subchannel and sample count, are added to the accumulated values in the measurement block for the subchannel. The start-subchannel count is the only measurement data that is updated in the measurement block if the channel-subsystem-timing facility is not available for the subchannel. (See “Channel-Subsystem Monitoring” on page 17-1 for more information.)

If a measurement-check condition is detected during the measurement-block update, the channel program is terminated at the subchannel. The subchannel is made status pending with primary, secondary, and alert status. Deferred condition code one is set, and the start-pending bit remains set to one. The device-status and subchannel-status fields are set to zero, and one of the measurement-check conditions is indicated in the extended-status flags of the format-0 ESW. The subchannel is not placed in the suspended state. (See “Subchannel-Control Field” on page 16-11)

4. The subchannel is placed in the suspended state.
5. If the subchannel is not resume pending at this point, the intermediate interruption condition due to suspension is recognized if the suppress-suspended-interruption bit of the ORB is zero; otherwise, the resume function is performed.

When a valid CCW that contains an S flag validly set to one becomes the current CCW during command chaining and the resume-pending condition is recognized, the resume function is performed instead of the suspend function.

When the first CCW of a channel program contains an S flag validly set to one and the resume-pending condition is not recognized, the suspend function is performed and causes the following actions to occur in the order given:

1. Channel-program execution is suspended prior to the selection of the device.
2. The subchannel is set up such that the resume function can be performed when the subchannel is next recognized to be resume pending.
3. If the measurement-block-update mode is active and the subchannel is enabled for the mode, the SSCH+RSCH count is incremented, and the accrued function-pending time (a function of the setting of the timing-facility bit) is added to the accumulated value in the measurement block for the subchannel.

If a measurement-check condition is detected during the measurement-block update, the channel program is not started at the subchannel. The subchannel is made status pending with primary, secondary, and alert status. Deferred condition code one is set, and the start-pending bit remains set to one. The device-status and subchannel-status fields are set to zero, and one of the measurement-check conditions is indicated in the extended-status flags of the format-0 ESW. The subchannel is not placed in the suspended state. (See “Subchannel-Control Field” on page 16-11)

4. The subchannel is placed in the suspended state.
5. If the subchannel is not resume pending at this point, the subchannel is made status
pending with intermediate status due to suspension if the suppress-suspended-interruption-control bit of the ORB is zero; otherwise, the resume function is performed.

When the first CCW of a channel program contains an S flag validly set to one and the resume-pending condition is recognized, the resume function is performed instead of the suspend function.

**Programming Notes:**

1. The execution of MODIFY SUBCHANNEL and START SUBCHANNEL completes with condition code 2 set if the designated subchannel is suspended. The start function is indicated at the subchannel while the subchannel is in the suspended state.

2. In certain situations, normal resumption of the execution of a channel program that has been suspended may not be desired. Normal termination of the suspended channel-program execution may be accomplished by:
   a. Executing HALT SUBCHANNEL and designating the subchannel.
   b. Modifying the CCWs in storage such that, when channel-program execution is resumed, the command transferred to the device is a control command with all modifier bits specified as zeros (no-operation) and with the chain-command flag specified as zero; and then executing RESUME SUBCHANNEL.
   c. When an IRB indicates measurement check along with zero device status, zero subchannel status, and status pending with primary, secondary, and alert status, it may indicate that the measurement check was detected during an attempt to place the subchannel into the suspended state.

3. If the suspended interruption is suppressed, the N condition and DCTI values applicable to the preceding subchannel-active period are not made available to the program. The execution of RESUME SUBCHANNEL when the subchannel is in the suspended state causes path-not-operational conditions and the N condition to be reset to zeros. Path-not-operational conditions and the N condition are not reset when RESUME SUBCHANNEL is executed and the designated subchannel is not in the suspended state.

**Commands and Flags**

Figure 15-7 lists the command codes for the seven commands and indicates which flags are defined for each command. Except for a format-1 CCW specifying transfer in channel, the flags are ignored for all commands for which they are not defined. The flags are reserved in a format-1 CCW specifying transfer in channel and must be zeros.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>M M M M M M 0</td>
<td>CD CC SLI PCI IDA S</td>
</tr>
<tr>
<td>Read</td>
<td>M M M M M M 1</td>
<td>CD CC SLI SK PCI IDA S</td>
</tr>
<tr>
<td>Read backward</td>
<td>M M M M 1 1 0</td>
<td>CD CC SLI SK PCI IDA S</td>
</tr>
<tr>
<td>Control</td>
<td>M M M M M M 1</td>
<td>CD CC SLI PCI IDA S</td>
</tr>
<tr>
<td>Sense</td>
<td>M M M M 0 1 0</td>
<td>CD CC SLI SK PCI IDA S</td>
</tr>
<tr>
<td>Sense ID</td>
<td>1 1 0 0 0 0 0</td>
<td>CD CC SLI SK PCI IDA S</td>
</tr>
<tr>
<td>Transfer in channel</td>
<td>X X X X 1 0 0 0</td>
<td>(See note below)</td>
</tr>
</tbody>
</table>

**Explanation:**

- **CC**: Chain command
- **CD**: Chain data
- **IDA**: Indirect data addressing
- **M**: Modifier bit
- **PCI**: Program-controlled interruption
- **S**: Suspend
- **SK**: Skip
- **SLI**: Suppress-length indication
- **X**: Ignored in a format-0 CCW; must be zero in a format-1 CCW

**Note:** Flags are ignored in a format-0 transfer-in-channel CCW and must be zeros in a format-1 transfer-in-channel CCW.

Figure 15-7. Command Codes and Flags

All flags have individual significance, except that the CC and SLI flags are ignored when the CD flag is set to one, and, for output forward operations the SK flag is ignored. The presence of the SLI flag is ignored for immediate operations involving format-0 CCWs, in which case the incorrect-length indication is suppressed regardless of the setting of the flag. The incorrect-length indication may be suppressed for immediate operations when executing a format-1 CCW, depending on the incorrect-length-suppression mode. The PCI flag is ignored during initial program loading. All flags, except the PCI flag, are ignored when the S flag is one.
Programming Notes:

1. A malfunction that affects the validity of data transferred in an I/O operation is signaled at the end of the operation by means of unit check or channel-data check, depending on whether the device (control unit) or the channel subsystem detected the error. In order to make use of the checking facilities provided in the system, data read in an input operation should not be used until the end of the operation has been reached and the validity of the data has been checked. Similarly, on writing, the copy of data in main storage should not be destroyed until the program has verified that no malfunction affecting the transfer and recording of data was detected.

2. An error condition may be recognized and the I/O operation terminated when 256 or more chained commands are executed with a device and none of the executed commands result in the transfer of any data. When this condition is recognized, program check is indicated.

3. All CCWs that require suppression of incorrect-length indications must use the SLI flag.

Branching in Channel Programs

The channel subsystem provides two methods to modify the normal sequential execution of the CCWs in a channel program. One is the transfer-in-channel (TIC) command (described in "Transfer in Channel"), which can be used to loop back to a previously executed CCW, or to connect contiguous segments of the channel program. The other method, which uses the status-modifier device-status bit (described in the publication \textit{ESA/390 Common I/O-Device Commands}, SA22-7204), allows conditions at the device to cause the channel to bypass the next CCW in the channel program.

Transfer in Channel

<table>
<thead>
<tr>
<th>Format-0 TIC CCW</th>
</tr>
</thead>
</table>
| \[
\begin{array}{c}
///
\end{array}
\begin{array}{c}
1000
\end{array}
\begin{array}{c}
CCW Address
\end{array}
\] |
| 0 | 8 | 31 |

<table>
<thead>
<tr>
<th>Format-1 TIC CCW</th>
</tr>
</thead>
</table>
| \[
\begin{array}{c}
/zerodot/zerodot/zerodot/zerodot1/zerodot/zerodot/zerodot
\end{array}
\begin{array}{c}
Zeros
\end{array}
\begin{array}{c}
CCW Address
\end{array}
\] |
| 00001000 | 32 | 63 |

The next CCW is fetched from the location in absolute main storage designated by the data-address field of the CCW specifying transfer in channel. The transfer-in-channel command does not initiate any I/O operation, and the I/O device is not signaled of the execution of the command. The purpose of the transfer-in-channel command is to provide chaining between CCWs not located in adjacent doubleword locations in an ascending order of addresses. The command can occur in both data and command chaining.

Bits 29-31 (format 0) or bits 61-63 (format 1) of a CCW that specifies the transfer-in-channel command must be zeros, designating a CCW on a doubleword boundary. Furthermore, a CCW specifying transfer in channel may not be fetched from a location designated by an immediately preceding transfer in channel. When either of these errors is detected or when an invalid address is designated in the transfer-in-channel command, the program-check condition is generated. When a CCW that specifies the transfer-in-channel command designates a CCW at a location protected against fetching, the protection-check condition is generated. Detection of these errors during data chaining causes the operation at the I/O device to be terminated and an interruption condition to be generated, while during command chaining it causes only an interruption condition to be generated.

The contents of the second half of the format-0 CCW, bit positions 32-63, are ignored. Similarly,
the contents of bit positions 0-3 of the format-0 CCW are ignored.

Bit positions 0-3 and 8-32 of the format-1 CCW must contain zeros; otherwise, a program-check condition is generated.

**Command Retry**

The channel subsystem has the capability to perform command retry, a procedure that causes a command to be retried without requiring an I/O interruption. This retry is initiated by the control unit presenting either of two status-bit combinations by means of a special sequence. When immediate retry can be performed, it presents a channel-end, unit-check, and status-modifier status-bit combination, together with device end. When immediate retry cannot be performed, the presentation of device end is delayed until the control unit is prepared. When device end is presented alone, the previous command is transferred again. If device end is accompanied by status modifier, command retry is not performed, and the channel subsystem command-chains to the CCW following the one for which command retry was signaled (for information on status modifier, see the publication *ESA/390 Common I/O-Device Commands*, SA22-7204). When the channel subsystem is not capable of performing command retry due to an error condition, or when any status bit other than device end or device end and status modifier accompanies the requested command-retry initiation, the retry is suppressed, and the subchannel becomes status pending. The SCSW stored by TEST SUBCHANNEL contains the status provided by the I/O device.

**Programming Note:** The following possible results of a command retry must be anticipated by the program:

1. A CCW containing a PCI may, if retried because of command retry, cause multiple PCI interruptions to occur.
2. If a CCW used in an operation is changed before that operation has been successfully completed, the results are unpredictable.

**Concluding I/O Operations before Initiation**

Subsequent to the execution of START SUBCHANNEL or RESUME SUBCHANNEL and before the first command is accepted, the start function can be ended at the subchannel by CANCEL SUBCHANNEL (if the instruction is installed), CLEAR SUBCHANNEL, or HALT SUBCHANNEL. If the I/O operation is ended by CANCEL SUBCHANNEL, there is no subsequent interruption condition from the I/O operation, and the subchannel is available for the initiation of another start function. However, the device may have signaled a busy condition while the canceled operation was start pending. In this case, the device owes a no-longer-busy signal to the channel subsystem. This may result in unsolicited device-end status before the next operation is initiated at the device. (See also “Clear Function” on page 15-14 and “Halt Function” on page 15-15.)

**Concluding I/O Operations during Initiation**

After the designated subchannel has been determined to be in a state such that START SUBCHANNEL can be executed, certain tests are performed on the validity of the information specified by the program and on the logical availability of the associated device. This testing occurs during or subsequent to the execution of START SUBCHANNEL and during command chaining and command retry.

A data-transfer operation is initiated at the subchannel and device only when no programming or equipment errors are detected by the channel subsystem and when the device responds with zero status during the initiation sequence. When the channel subsystem detects or the device signals any unusual condition during the initiation of an I/O operation, the command is said to be not accepted. In this case, the subchannel becomes status pending with primary, secondary, and alert status. Deferred condition code 1 is set, and the start-pending bit remains set to one.

Conditions that preclude the initiation of an I/O operation are detailed in the SCSW stored by TEST SUBCHANNEL. In this situation, the device is not started, no interruption conditions are generated subsequent to TEST SUBCHANNEL, and the
subchannel is idle. The device is immediately available for the initiation of another operation, provided the command was not rejected because of the busy or not-operational condition.

When an unusual condition causes a command to be not accepted during the initiation of an I/O operation by command chaining or command retry, an interruption condition is generated, and the subchannel becomes status pending with combinations of primary, secondary, and alert status as a function of the status signaled by the device. The status describing the condition remains at the subchannel until cleared by TEST SUBCHANNEL. The conditions are indicated to the program by means of the corresponding status bits in the SCSW. A path-not-operational condition recognized during command chaining is signaled to the program by means of an interface-control-check indication. The new I/O operation at the device is not started.

START SUBCHANNEL is executed independent of its associated device. Tests on most program-specified information, on device availability and unit status, and on most error conditions are performed subsequent to the execution of START SUBCHANNEL. When any conditions are detected that preclude the performance of the start function, an interruption condition is generated by the channel subsystem and placed at the subchannel, causing it to become status pending.

### Immediate Conclusion of I/O Operations

During the initiation of an I/O operation, the device can accept the command and signal the channel-end condition immediately upon receipt of the command code. An I/O operation causing the channel-end condition to be signaled during the initiation sequence is called an immediate operation. Status generated by the device for the immediate command, when command chaining is not specified and command retry is not signaled, causes the subchannel to become status pending with combinations of primary, secondary, intermediate, and alert status as a result of information specified in the ORB and CCW and status presented by the device. If the immediate operation is the first operation of the channel program, deferred condition code 1 is set and accompanies the status indications. If intermediate status is indicated, the indication can occur only as a result of the CCW having the PCI flag set to one (see "Program-Controlled Interruption" on page 15-35).

Whenever command chaining is specified after an immediate operation and no unusual conditions have been detected during the operation, or when command retry occurs for an immediate operation, an interruption condition is not generated. The subsequent commands in the chain are handled normally, and, usually, the channel-end condition for the last CCW generates a primary interruption condition. If device end is signaled with channel end, a secondary interruption condition is also generated.

Whenever immediate completion of an I/O operation is signaled, no data has been transferred to or from the device, and the data address in the CCW is not checked for validity. If the subchannel is in the incorrect-length-suppression mode, incorrect length is not indicated to the program, and command chaining is performed when specified. If the subchannel is in the incorrect-length-indication mode, incorrect length and command chaining are under control of the SLI and chain-command flags. The conditions that cause the incorrect-length indication to be suppressed are summarized in Figure 15-6 on page 15-32.

**Programming Note:** I/O operations for which the entire operation is specified in the command code may be performed as immediate operations. Whether the command is executed as an immediate operation depends on the operation and type of device.

### Concluding I/O Operations during Data Transfer

When the subchannel has been passed the contents of an ORB, the subchannel is said to be start pending. When the I/O operation has been initiated and the command has been accepted, the subchannel becomes subchannel-and-device active and remains in that state unless (1) the channel subsystem detects an equipment malfunction, (2) the operation is concluded by the execution of CLEAR SUBCHANNEL or HALT SUBCHANNEL, or (3) status that causes a primary interruption condition to be recognized (usually channel end) is accepted from the device. When command chaining and command retry are not
specified or when chaining is suppressed because of unusual conditions, the status that is recognized as primary status causes the operation at the sub-channel to be concluded and an interruption condition to be generated. The status bits in the associated SCSW indicate primary status and the unusual conditions, if any. The device can present status that is recognized as primary status at any time after the initiation of the I/O operation, and the presentation of status may occur before any data has been transferred.

For operations not involving data transfer, the device normally controls the timing of the channel-end condition. The duration of data-transfer operations may be variable and may be controlled by the device or the channel subsystem.

Excluding equipment errors and the execution of the CLEAR SUBCHANNEL, HALT SUBCHANNEL, and RESET CHANNEL PATH instructions, the channel subsystem signals the device to conclude the performance of an I/O operation during data transfer whenever any of the following conditions occurs:

- The storage areas designated for the operation are exhausted or filled.
- A program-check condition is detected.
- A protection-check condition is detected.
- A chaining-check condition is detected.
- A channel-control-check condition is detected that does not affect the control of the I/O operation.

The first of these conditions occurs when the channel subsystem has decremented the count to zero in the last CCW associated with the operation. A count of zero indicates that the channel subsystem has transferred all information specified by the I/O operation. The other four conditions are due to errors and cause premature conclusion of data transfer. In either case, the conclusion is signaled in response to a service request from the device and causes data transfer to cease. If the device has no blocks defined for the operation (such as writing on magnetic tape), it concludes the operation and presents channel-end status.

The device can control the duration of an operation and the timing of channel end by blocking of data. On certain operations for which blocks are defined (such as reading on magnetic tape), the device does not present channel-end status until the end of the block is reached, regardless of whether the device has been previously signaled to conclude data transfer.

Checking for the validity of the data address is performed only as data is transferred to or from main storage. When the initial data address in the CCW is invalid, no data is transferred during the operation, and the device is signaled to conclude the operation in response to the first service request. On writing, devices such as magnetic-tape units request the first byte of data before any mechanical motion is started, and, if the initial data address is invalid, the operation is terminated by the channel subsystem before the recording medium has been advanced. However, since the operation has been initiated at the device, the device presents channel-end status, causing the channel subsystem to recognize a primary interruption condition. Subsequently, the device also presents device-end status, causing the channel subsystem to recognize a secondary interruption condition. Whether a block at the device is advanced when no data is transferred depends on the type of device.

When command chaining takes place, the sub-channel is in the subchannel-and-device-active state from the time the first I/O operation is initiated at the device until the device presents channel-end status for the last I/O operation of the chain. The subchannel remains in the device-active state until the device presents the device-end status for the last I/O operation of the chain.

Any unusual conditions cause command chaining to be suppressed and a primary interruption condition to be generated. The unusual conditions can be detected by either the channel subsystem or the device, and the device can provide the indications with channel end, control unit end, or device end. When the channel subsystem is aware of the unusual condition by the time the channel-end status for the operation is accepted, the chain is ended as if the operation during which the condition occurred were the last operation of the chain. The device-end status is recognized as a secondary interruption condition whether presented together with the channel-end status or separately. If the device presents unit check or unit exception together with either control unit end
or device end as status that causes the channel subsystem to recognize the primary interruption condition, then the subchannel-and-device-active state of the subchannel is terminated, and the subchannel is made status pending with primary, secondary, and alert status. Intermediate status may also be indicated if an intermediate interruption condition previously existed at the subchannel for the initial-status-interruption condition or the PCI condition and that condition still remains pending at the subchannel. The channel-end status that was presented to the channel subsystem previously when command chaining was signaled is not made available to the program.

Channel-Path-Reset Function

Subsequent to the execution of RESET CHANNEL PATH, the channel-path-reset function is performed. The performance of the function consists of: (1) issuing the reset signal on the designated channel path and (2) causing a channel report to be made pending, indicating the completion of the channel-path-reset function.

Channel-Path-Reset-Function Signaling

The channel subsystem issues the reset signal on the designated channel path. As part of this operation, the following actions are taken:

1. All internal indications associated with control-unit-busy, device-busy, and allegiance conditions for the designated channel path are reset. These indications are reset at all subchannels that have access to the designated channel path. The reset function has no other effect on subchannels, including those having I/O operations in progress.

2. If the channel path fails to respond properly to the reset signal (see \textit{I/O-System Reset\textsuperscript{\textbackslash page 17-13}} for a detailed description) or, because of a malfunction, the reset signal could not be issued, the channel path is made physically not available at each applicable subchannel.

3. If an I/O operation is in progress at the device and the device is actively communicating on the channel path in the performance of that I/O operation when the reset signal is received on that channel path, the I/O operation is reset, and the control unit and device immediately terminate current communication with the channel subsystem. (To avoid possible misinterpretation of unsolicited device-end status, programming measures can be taken as described in programming note \textsuperscript{2 on page 15-46}.)

4. If an I/O operation is in progress in the multipath mode at the device and the device is not currently communicating over the channel path in the performance of that I/O operation when the reset signal is received, then the I/O operation may or may not be reset depending on whether another channel path is available for selection in the same multipath group for the device. If there is at least one other channel path in the multipath group for the device that is available for selection, the I/O operation is not reset. However, the channel path on which the system reset is received is removed from the current set of channel paths that form the multipath group. If the channel path on which the reset signal is received is the only channel path of a multipath group, or if the device is operating in the single-path mode, the I/O operation is reset.

5. The channel-path-reset function causes I/O operations to be terminated at the device as described above; however, I/O operations are never terminated at the subchannel by the channel-path-reset function.

If an I/O operation is in progress at the subchannel and the channel path designated for the performance of the channel-path-reset function is being used for that I/O operation, the subchannel may or may not accurately reflect the progress of the I/O operation up to that instant. The subchannel remains in the state that exists at the time the channel-path-reset function is performed until the state is changed because of some action taken by the program or by the device.

Channel-Path-Reset-Function-Completion Signaling

After the reset signal has been issued and an attempt has been made to issue the reset signal, or after it has been determined that the reset signal cannot be issued, the channel-path-reset function is completed. (See \textit{Reset Signal\textsuperscript{\textbackslash page 17-13}}.)
As a result of the channel-path-reset function being performed, a channel report is made pending (see "Channel-Subsystem Recovery" on page 17-24) to report the results. If the channel path responds properly to the system-reset signal, the channel report indicates that the channel path has been initialized and is physically available for use. If the reset signal was issued but either the channel path failed to respond properly or the channel path was already not physically available at each subchannel having access to the channel path, the channel report indicates that the channel path has been initialized but is not physically available for use. If, because of a malfunction or because the designated channel path is not in the configuration, the reset signal could not be issued, the channel report indicates that the channel path has not been initialized and is not physically available for use.

Programming Notes:

1. If an I/O operation is in progress in the multipath mode when the channel-path-reset function is performed on a channel path of the multipath group, it is possible for the I/O operation to be continued on a remaining channel path of the group.

2. When the performance of the channel-path-reset function causes the I/O operation at the device to be reset, unsolicited device-end status presented by the device, if any, may be erroneously interpreted by the channel subsystem to be chaining status and thus cause the channel subsystem to continue the chain of commands. If this situation occurs, then the device-end status is not made available to the program, and the device is selected again by the channel subsystem; however, the device may interpret the initiation sequence as the beginning of a new channel program instead of as command chaining. This possibility can be avoided by issuing CLEAR SUBCHANNEL or HALT SUBCHANNEL, designating the affected subchannels, prior to issuing RESET CHANNEL PATH.

3. The performance of the channel-path-reset function may, on some models, cause overruns to occur on other channel paths.

4. Even though reset is signaled on the designated channel path, allegiances to that channel path by one or more devices may not have been reset because of a malfunction at a control unit or a malfunction at the physical channel path to the control unit.
Chapter 16. I/O Interruptions

Interruption Conditions . . . . . . . . . . . . . 16-2
Intermediate Interruption Condition . . . . . . . 16-4
Primary Interruption Condition . . . . . . . . . . 16-4
Secondary Interruption Condition . . . . . . . . 16-4
Alert Interruption Condition . . . . . . . . . . 16-4
Priority of Interruptions . . . . . . . . . . . . . 16-4

Interruption Action . . . . . . . . . . . . . . . . 16-5
 Interruption-Response Block . . . . . . . . . . . . 16-6
Subchannel-Status Word . . . . . . . . . . . . . 16-6
 Subchannel Key . . . . . . . . . . . . . . 16-8
Suspend Control (S) . . . . . . . . . . . . . . 16-8
Extended-Status-Word Format (L) . . . . . . . . 16-8
Deferred Condition Code (CC) . . . . . . . . . 16-8
Format (F) . . . . . . . . . . . . . . . . 16-10
Prefetch (P) . . . . . . . . . . . . . . . . 16-10
Initial-Status-Interruption Control (I) . . . . 16-11
Address-Limit-Checking Control (A) . . . . . . -11
 Suppress-Suspended Interruption (U) . . . . . . 16-11
Subchannel-Control Field . . . . . . . . . . . . 16-11
 Zero Condition Code (Z) . . . . . . . . . . . . 16-11
Extended Control (E) . . . . . . . . . . . . . 16-11
Path Not Operational (N) . . . . . . . . . . . . 16-12
Function Control (FC) . . . . . . . . . . . . . 16-12
Activity Control (AC) . . . . . . . . . . . . . 16-13
Status Control (SC) . . . . . . . . . . . . . . 16-16

CCW-Address Field . . . . . . . . . . . . . . . . 16-18
Device-Status Field . . . . . . . . . . . . . . . . 16-23
Subchannel-Status Field . . . . . . . . . . . . . 16-23
Program-Controlled Interruption . . . . . . . . . . 16-23
Incorrect Length . . . . . . . . . . . . . . . . 16-23
Program Check . . . . . . . . . . . . . . . . 16-24
Protection Check . . . . . . . . . . . . . . . . 16-26
Channel-Data Check . . . . . . . . . . . . . . . 16-26
Channel-Control Check . . . . . . . . . . . . . 16-27
Interface-Control Check . . . . . . . . . . . . . 16-28
Chaining Check . . . . . . . . . . . . . . . . 16-29
Count Field . . . . . . . . . . . . . . . . . . . 16-29
Extended-Status Word . . . . . . . . . . . . . . 16-32
Extended-Status Format 0 . . . . . . . . . . . . . 16-32
Subchannel Logout . . . . . . . . . . . . . . . . 16-32
Extended-Report Word . . . . . . . . . . . . . . 16-36
Failing-Storage Address . . . . . . . . . . . . . . 16-38
Extended-Subchannel-Logout
 Descriptor (ESLD) . . . . . . . . . . . . . . . . 16-38
Secondary-CCW Address . . . . . . . . . . . . . . 16-38
Extended-Status Format 1 . . . . . . . . . . . . 16-38
Extended-Status Format 2 . . . . . . . . . . . . . 16-39
Extended-Status Format 3 . . . . . . . . . . . . . 16-40
Extended-Control Word . . . . . . . . . . . . . . . . . 16-40
Extended-Measurement Word . . . . . . . . . . . . 16-41

When an I/O operation or sequence of I/O operations initiated by the execution of START SUB-CHANNEL is ended, the channel subsystem and the device generate status conditions. The generation of these conditions can be brought to the attention of the program by means of an I/O interruption or by means of the execution of the TEST PENDING INTERRUPTION instruction. (During certain abnormal situations, these conditions can be brought to the attention of the program by means of a machine-check interruption. See "Channel-Subsystem Recovery" on page 17-24 for details.) The status conditions, as well as an address and a count indicating the extent of the operation sequence, are presented to the program in the form of a subchannel-status word (SCSW). The SCSW is stored in an interruption-response block (IRB) during the execution of TEST SUB-CHANNEL.

Normally an I/O operation is being performed until the device signals primary interruption status. Primary interruption status can be signaled during initiation of an I/O operation, or later. An I/O operation can be terminated by the channel subsystem performing a clear or halt function when it detects an equipment malfunction, a program check, a chaining check, a protection check, or an incorrect-length condition, or by performing a clear, halt, or channel-path-reset function as a result of the execution of CLEAR SUBCHANNEL, HALT SUBCHANNEL, or RESET CHANNEL PATH, respectively.

I/O interruptions provide a means for the CPU to change its state in response to conditions that occur at I/O devices or subchannels. These conditions can be caused by the program, by the channel subsystem, or by an external event at the device.
**Interruption Conditions**

The conditions causing requests for I/O interruptions to be initiated are called I/O-interruption conditions. When an interruption condition is recognized by the channel subsystem, it is indicated at the appropriate subchannel. The subchannel is then said to be status pending. The subchannel becoming status pending causes the channel subsystem to generate an I/O-interruption request. An I/O-interruption request can be brought to the attention of the program only once.

An I/O-interruption request remains pending until it is accepted by a CPU in the configuration, is withdrawn by the channel subsystem, or is cleared by means of the execution of TEST PENDING INTERRUPTION, TEST SUBCHANNEL, or CLEAR SUBCHANNEL, or by means of subsystem reset. When a CPU accepts an interruption request and stores the associated interruption code, the interruption request is cleared. Alternatively, an I/O-interruption request can be cleared by means of the execution of TEST PENDING INTERRUPTION. In either case, the subchannel remains status pending until the associated interruption condition is cleared when TEST SUBCHANNEL or CLEAR SUBCHANNEL is executed or when the subchannel is reset.

An I/O-interruption condition is normally cleared by means of the execution of TEST SUBCHANNEL. If TEST SUBCHANNEL is executed, designating a subchannel that has an I/O-interruption request pending, both the interruption request and the interruption condition at the subchannel are cleared. The interruption request and the interruption condition can also be cleared by CLEAR SUBCHANNEL.

A device-end status condition generated by the I/O device and presented following the conclusion of the last I/O operation of a start function is reset at the subchannel by the channel subsystem without generating an I/O-interruption condition or I/O-interruption request if the subchannel is currently start pending and if the status contains device end either alone or accompanied by control unit end. If any other status bits accompany the device-end status bit, then the channel subsystem generates an I/O-interruption request with deferred condition code 1 indicated.

When an I/O operation is terminated because of an unusual condition detected by the channel subsystem during the command-initiation sequence, status describing the interruption condition is placed at the subchannel, causing it to become status pending. If the unusual condition is detected by the device, the device-status field of the associated SCSW identifies the condition.

When command chaining takes place, the generation of status by the device does not cause an interruption, and the status is not made available to the program.

When the channel subsystem detects any of the following interruption conditions, it initiates a request for an I/O interruption without necessarily communicating with, or having received the status byte from, the device:

- A programming error associated with the contents of the ORB passed to the subchannel by the previous execution of START SUBCHANNEL
- A suspend flag set to one in the first CCW fetched that initiates channel-program execution for either START SUBCHANNEL or RESUME SUBCHANNEL, and suppress suspended interruption not specified in the ORB
- A programming error associated with the first CCW or first IDAW

These interruption conditions from the subchannel, except for the suspended condition, can be accompanied by other subchannel-status indications, but the device-status indications are all stored as zeros.

The channel subsystem issues the clear signal to the device when status containing unit check is presented to a subchannel that is disabled or when the device is not associated with any subchannel. However, if the presented status does not contain unit check, the status is accepted by the channel subsystem and discarded without causing the subchannel to become status pending.

An interruption condition caused by the device may be accompanied by multiple device-status conditions. Furthermore, more than one interruption condition associated with the same device can be accepted by the channel subsystem without an intervening I/O interruption. As an
example, when the channel-end condition is not cleared at the device by the time device end is generated, both conditions may be cleared at the device concurrently and indicated in the SCSW together. Alternatively, channel-end status may have been previously accepted at the subchannel, and an I/O interruption may have occurred; however, the associated status-pending condition may not have been cleared by TEST SUB-CHANNEL by the time device-end status was accepted at the subchannel. In this situation, the device-end status may be merged with the channel-end status without causing an additional I/O interruption. Whether an interruption condition may be merged at the subchannel with other existing interruption conditions depends upon whether the interruption condition is unsolicited or solicited.

**Unsolicited Interruption Condition:** An unsolicited interruption condition is any interruption condition that is unrelated to the performance of a clear, halt, resume, or start function. An unsolicited interruption condition is identified at the subchannel as alert status. An unsolicited interruption condition can be generated only when the subchannel is not device active.

The subchannel and device status associated with an unsolicited interruption condition is never merged with that of any currently existing interruption condition. If the subchannel is currently status pending, the unsolicited interruption condition is held in abeyance in either the channel sub-system or the device, as appropriate, until the status-pending condition has been cleared. Whenever the subchannel is idle and zero status is presented by the device, the status is discarded.

**Solicited Interruption Condition:** A solicited interruption condition is any interruption condition generated as a direct consequence of performing or attempting to perform a clear, halt, resume, or start function. Solicited interruption conditions include any interruption condition generated while the subchannel is either subchannel-and-device active or device active. The subchannel and device status associated with a solicited interruption condition may be merged at the subchannel with that of another currently existing solicited interruption condition. Figure 16-1 describes the interruption condition that results from any combination of bits in the status-control field of the SCSW.

<table>
<thead>
<tr>
<th>Status-Control Field</th>
<th>Status-Control-Bit Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alert</td>
<td>1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Primary</td>
<td>0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>Secondary</td>
<td>0 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0</td>
</tr>
<tr>
<td>Intermediate</td>
<td>0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>Status pending</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

| Resulting interrup- | E | S | S | S | S | S | S | S |
| tion condition      | - | S | S | S | S | S | S | S |

**Explanation:**

- Combination does not occur.
- E Unsolicited or solicited interruption condition.
- S Solicited interruption condition.
- 0 Indicates the bit stored as zero.
- 1 Indicates the bit stored as one.

*Figure 16-1. Interruption Condition for Status-Control-Bit Combinations*
Intermediate Interruption Condition

An intermediate interruption condition is a solicited interruption condition that indicates that an event has occurred for which the program had previously requested notification. An intermediate interruption condition is described by solicited subchannel status, the Z bit, the subchannel-suspended condition, or any combination of the three. An intermediate interruption condition can occur only after it has been requested by the program through the use of flags in the ORB or a CCW. Depending on the state of the subchannel, the performance or suspension of the I/O operation continues, unaffected by the setting of the intermediate-status bit.

An intermediate interruption condition can be indicated only together with one of the following indications:

1. Subchannel active
2. Status pending with primary status alone
3. Status pending with primary status together with alert status or secondary status or both
4. Suspended

If only the intermediate-status bit and the status-pending bit of the status-control field are ones during the execution of TEST SUBCHANNEL, the device-status field is zero.

Primary Interruption Condition

A primary interruption condition is a solicited interruption condition that indicates the performance of the start function is completed at the subchannel. A primary interruption condition is described by the SCSW stored as a result of the execution of TEST SUBCHANNEL while the subchannel is status pending with primary status. Once the channel subsystem has accepted status from the device that causes a secondary interruption condition to be recognized, the start function is completed at the device.

Secondary Interruption Condition

A secondary interruption condition is a solicited interruption condition that normally indicates the completion of an I/O operation at the device. A secondary interruption condition is also generated by the channel subsystem if the start function is terminated because a solicited alert interruption condition is recognized prior to initiating the first I/O operation at the device. A secondary interruption condition is described by the SCSW stored as a result of the execution of TEST SUBCHANNEL while the subchannel is status pending with secondary status. Once the channel subsystem has accepted status from the device that causes a secondary interruption condition to be recognized, the start function is completed at the device.

Alert Interruption Condition

An alert interruption condition is either a solicited interruption condition that indicates the occurrence of an unusual condition in a halt, resume, or start function or an unsolicited interruption condition that describes a condition unrelated to the performance of a halt, resume, or start function. An alert interruption condition is described by the SCSW stored as a result of the execution of TEST SUBCHANNEL while the subchannel is status pending with alert status. An alert interruption condition may be generated by either the channel subsystem or the device. Nonzero alert status is always brought to the attention of the program.

Priority of Interruptions

All requests for an I/O interruption are asynchronous to any activity in any CPU, and interruption requests associated with more than one subchannel can exist at the same time. The priority of interruptions is controlled by two types of mechanisms — one establishes within the channel subsystem the priority among interruption requests from subchannels associated with the same I/O-interruption subclass, and another establishes within a given CPU the priority among requests from subchannels of different I/O-interruption sub-
classes. The channel subsystem requests an I/O interruption only after it has established priority among requests from its subchannels. The conditions responsible for the I/O-interruption requests associated with subchannels are preserved at the subchannels until cleared by a CPU's execution of TEST SUBCHANNEL or CLEAR SUBCHANNEL or I/O-system reset is performed.

The assignment of priority among requests for interruption from subchannels of the same I/O-interruption subclass is in the order that the need for interruption is recognized by the channel subsystem. The order of recognition by the channel subsystem is a function of the type of interruption condition and the type of channel path. For the parallel-I/O-interface type of channel path, the order depends on the electrical position of the device on the channel path to which it is attached. (A device's electrical position on the parallel-I/O interface is not related to its device address.)

The assignment of priority among requests for interruption from subchannels of different I/O-interruption subclasses is made by the CPU according to the numerical value of the I/O-interruption subclass codes (with zero having highest priority), in conjunction with the I/O-interruption-subclass mask in control register 6. The numerical value of the I/O-interruption-subclass code directly corresponds to the bit position in the I/O-interruption-subclass mask in control register 6 of a CPU. If, in any CPU, an I/O-interruption-subclass-mask bit is zero, then all subchannels having an I/O-interruption-subclass code numerically equal to the associated position in the mask register are said to be masked off in the respective CPU. Therefore, a CPU accepts the highest-priority I/O-interruption request from a subchannel that has the lowest-numbered I/O-interruption subclass code that is not masked off by a corresponding bit in control register 6 of that CPU. When the highest-priority interruption request is accepted by a CPU, it is cleared so that the interruption request is not accepted by any other CPU in the configuration.

The priority of interruption handling can be modified by the execution of either TEST SUBCHANNEL or CLEAR SUBCHANNEL. When either of these instructions is executed and the designated subchannel has an interruption request pending, that interruption request is cleared, without regard to any previous established priority. The relative priority of the remaining interruption requests is unchanged.

**Programming Notes:**

1. The I/O-interruption subclass mask is in control register 6, which has the following format:

   ![Control Register Format]

<table>
<thead>
<tr>
<th>ISC Mask</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 32 40 63</td>
<td></td>
</tr>
</tbody>
</table>

2. Control register 6 is set to all zeros during initial CPU reset.

**Interruption Action**

An I/O interruption can occur only when the I/O-interruption-subclass-mask bit associated with the subchannel is one and the CPU is enabled for I/O interruptions.

The interruption occurs at the completion of a unit of operation (see "Point of Interruption" on page 5-21). If the channel subsystem establishes the priority among requests for interruption from subchannels while the CPU is disabled for I/O interruptions, the interruption occurs immediately after the completion of the instruction enabling the CPU and before the next instruction is executed, provided that the I/O-interruption-subclass-mask bit associated with the subchannel is one. Alternatively, if the channel subsystem establishes the priority among requests for interruption from subchannels while the I/O-interruption-subclass-mask bit is zero for each subchannel that is status pending, the interruption occurs immediately after the completion of the instruction that sets at least one of the I/O-interruption-subclass-mask bits to one, provided that the CPU is also enabled for I/O interruptions. This interruption is associated with the highest-priority I/O-interruption request, as established by the CPU.

If the channel subsystem has not established the priority among requests for interruption from the subchannels by the time the interruption is allowed, the interruption does not necessarily occur immediately after the completion of the instruction enabling the CPU. A delay can occur...
regardless of how long the interruption condition has existed at the subchannel.

The interruption causes the current PSW to be stored as the input/output old PSW at real locations 368-383 and causes the I/O-interruption code associated with the interruption to be stored at real locations 184-195 of the CPU allowing the interruption. Subsequently, a new input/output PSW is loaded from real locations 496-511, and processing resumes in the CPU state indicated by that PSW. The subchannel causing the interruption is identified by the interruption code.

The I/O-interruption code has the following format when it is stored. The code is described in "TEST PENDING INTERRUPTION" on page 14-18.

<table>
<thead>
<tr>
<th>Hex. Dec.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 184</td>
<td>Subsystem-Identification Word</td>
</tr>
<tr>
<td>BC 188</td>
<td>I/O-Interruption Parameter</td>
</tr>
<tr>
<td>C0 192</td>
<td>I/O-Interruption-Identification Word</td>
</tr>
</tbody>
</table>

Programming Note: The I/O-interruption subclass code for all subchannels is set to zero by I/O-system reset. It may be set to any of the values 0-7 by the execution of MODIFY SUBCHANNEL. (The operation of the instruction is described in "MODIFY SUBCHANNEL" on page 14-7.)

The length of the subchannel-status and extended-status words is 12 bytes and 20 bytes, respectively. The length of the extended-control word is 32 bytes. When the extended-control bit, bit 14 of word 0 of the SCSW, is zero, words 8-15 of the interruption-response block may or may not be stored. The length of the extended-measurement word is 32 bytes. When the conditions for storing the extended-measurement word are not met (see "Extended-Measurement Word" on page 16-41), words 16-23 of the interruption-response block may or may not be stored.

**Interruption-Response Block**

The interruption-response block (IRB) is the operand of TEST SUBCHANNEL. The two rightmost bits of the IRB address are zeros, designating the IRB on a word boundary. The IRB contains three major fields: the subchannel-status word, the extended-status word, and the extended-control word. When the extended-I/O-measurement-word mode is enabled at the subchannel, the IRB contains a fourth major field, the extended-measurement word. The format of the IRB is as follows:

```
Word 0 Subchannel-Status Word
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
```

The length of the subchannel-status and extended-status words is 12 bytes and 20 bytes, respectively. The length of the extended-control word is 32 bytes. When the extended-control bit, bit 14 of word 0 of the SCSW, is zero, words 8-15 of the interruption-response block may or may not be stored. The length of the extended-measurement word is 32 bytes. When the conditions for storing the extended-measurement word are not met (see "Extended-Measurement Word" on page 16-41), words 16-23 of the interruption-response block may or may not be stored.

**Subchannel-Status Word**

The subchannel-status word (SCSW) provides to the program indications describing the status of a subchannel and its associated device. If performance of a halt, resume, or start function has occurred, the SCSW may describe the conditions under which the operation was concluded.

The SCSW is stored when TEST SUBCHANNEL is executed and the designated subchannel is operational. The SCSW is placed in words 0-2 of the IRB that is designated as the TEST SUBCHANNEL operand. When STORE SUBCHANNEL is executed, the SCSW is stored in words 7-9 of the subchannel-information block (described in "Subchannel-Information Block" on page 15-1). Figure 16-2 on page 16-7 shows the format of the SCSW and summarizes its contents.
The contents of the subchannel-status word (SCSW) depend on the state of the subchannel when the SCSW is stored. Depending on the state of the subchannel and the device, the specific fields of the SCSW may contain (1) information pertaining to the last operation, (2) information unrelated to the performance of an operation, (3) zeros, or (4) meaningless values. The following descriptions indicate when an SCSW field contains meaningful information.
Subchannel Key
When the start-function bit, bit 17 of word 0, is one, bit positions 0-3 of word 0 contain the access key used during performance of the associated start function. These bits are identical with the key specified in bit positions 0-3 of word 1 of the ORB. The subchannel key is meaningful only when the start-function bit, bit 17 of word 0, is one.

Suspend Control (S)
When the start-function bit, bit 17 of word 0, is one, bit 4 of word 0, when one, indicates that the suspend function can be initiated at the subchannel. Bit 4 is meaningful only when bit 17 is one. If bit 17 is one and bit 4 is one, channel-program execution can be suspended if the channel subsystem recognizes an S flag set to one in a CCW. If bit 4 is zero, channel-program execution cannot be suspended, and, if an S flag set to one in a CCW is encountered, a program-check condition is recognized.

Extended-Status-Word Format (L)
When the status-pending bit, bit 31 of word 0, is one, bit 5 of word 0, when one, indicates that a format-0 ESW has been stored. A format-0 ESW is stored when an interruption condition containing any of the following indications is cleared by TEST SUBCHANNEL:

- Channel-data check
- Channel-control check
- Interface-control check
- Measurement-block-program check
- Measurement-block-data check
- Measurement-block-protection check
- Path verification required
- Authorization check
- Extended subchannel logout pending

The deferred condition code is meaningful when the subchannel is status pending with any combination of status and only when the start-function bit of the function-control field in the SCSW is one. The meaning of the deferred condition code for each value when the subchannel is status pending is given in Figure 16-3 on page 16-10.

Deferred Condition Code 0: A normal I/O interruption has taken place.

Deferred Condition Code 1: Status is present in the SCSW that was presented by the associated device or generated by the channel subsystem subsequent to the setting of condition code 0 for START SUBCHANNEL or RESUME SUBCHANNEL. If only the alert-status bit and the status-pending bit of the status-control field of the SCSW are ones, the status present is not related to the execution of a channel program. If the intermediate-status bit, the primary-status bit, or both are ones, then the status is related to the execution of the channel program specified by the most recently executed START SUBCHANNEL instruction or implied by the most recently executed RESUME SUBCHANNEL instruction. (See “Immediate Conclusion of I/O Operations” on page 15-43) If the secondary-status bit is one and the primary-status bit is zero, the status present is related to the channel program specified by the START SUBCHANNEL instruction or implied by the RESUME SUBCHANNEL instruction that preceded the most recently executed START SUBCHANNEL instruction.

Deferred Condition Code 2: This code does not occur and is reserved for future use.

Deferred Condition Code 3: An attempted device selection has occurred, and the device appeared not operational on all of the channel paths that were available for selection of the device.

A device appears not operational when it does not respond to a selection attempt by the channel subsystem. This occurs when the control unit is not provided in the system, when power is off in...
the control unit, or when the control unit has been logically switched off the channel path. The not-operational state is also indicated when the control unit is provided and is capable of attaching the device, but the device has not been installed and the control unit is not designed to recognize the device being selected as one of its attached devices. (See also “I/O Addressing” on page 13-5.)

A deferred condition code 3 also can be set by the channel subsystem if no channel paths to the device are available for selection. (See Figure 16-3 on page 16-10.)

**Programming Notes:**

1. If, during performance of a start function, the I/O device being selected is not installed or has been logically removed from the control unit, but the associated control unit is operational and the control unit recognizes the I/O device being selected as one of its I/O devices, the control unit, depending upon the model, either fails to recognize the address of the I/O device or considers the I/O device to be not ready. In the former case, a path-not-operational condition is recognized, subject to the setting of the path-operational mask. (See “Path-Operational Mask (POM)” on page 15-6.) In the latter case, the not-ready condition is indicated when the control unit responds to the selection and indicates unit check whenever the not-ready state precludes successful initiation of the operation at the I/O device. In this case, unit-check status is indicated in the SCSW, the subchannel becomes status pending with primary, secondary, and alert status, and with deferred condition code 1 indicated. (See the publication *ESA/390 Common I/O-Device Commands*, SA22-7204, for a description of unit-check status.) Refer to the System Library publication for the control unit to determine how the condition is indicated.

2. The deferred condition code is 1, and the status-control field contains the status-pending and intermediate-status bits or the status-pending, intermediate-status, and alert-status bits as ones when HALT SUBCHANNEL has been executed and the designated subchannel is suspended and status pending with intermediate status. If the alert-status bit is one, then subchannel-logout information was generated as a result of attempting to issue the halt signal to the device.
<table>
<thead>
<tr>
<th>Bit 6</th>
<th>Bit 7</th>
<th>Status Control</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>AIPSX</td>
<td>Normal I/O interruption</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIP-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A-PSX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A-P-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I-PSX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I-P-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I--X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>---PSX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--P-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>---SX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--PSX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--SX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--P-X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>AIPSX</td>
<td>Either an immediate operation, with chaining not specified, has</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIP-X</td>
<td>ended normally, or the setting of some status condition precluded the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AIP-X</td>
<td>initiation or resumption of a requested I/O operation at the device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A-P-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A--SX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A---X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-IPSX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-IP-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-I--X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--PSX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--P-X</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>---SX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>--PX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>---SX</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>----X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-PSX</td>
<td>The device is not operational on any available path or, if a dedicated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I-PSX</td>
<td>allegiance condition exists, the device is not operational on the path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>to which the dedicated allegiance is owed.</td>
</tr>
</tbody>
</table>

Explanation:

- Bit is zero.
- The allowed combinations of status-control-bit settings when the start-function bit is one in the function-control field.
- The condition is encountered after the execution of HALT SUBCHANNEL when the subchannel is currently suspended.
- The condition is encountered after the execution of HALT SUBCHANNEL when the subchannel is currently start pending.
- A Alert status.
- I Intermediate status.
- P Primary status.
- S Secondary status.
- X Status pending.

Figure 16-3. Deferred-Condition-Code Meaning for Status-Pending Subchannel

**Format (F)**

When the start-function bit, bit 17 of word 0, is one, bit 8 of word 0 indicates the format of the CCWs associated with an I/O operation. The format bit is meaningful only when bit 17 is one. If bit 8 of word 0 is zero, format-0 CCWs are indicated. If it is one, format-1 CCWs are indicated. (See "Channel-Command Word" on page 15-27 for the description of the two CCW formats.)

**Prefetch (P)**

When the start-function bit, bit 17 of word 0, is one, bit 9 of word 0 indicates whether or not unlimited prefetching of CCWs, IDAWs, and associated data is allowed. The prefetch bit is meaningful only when bit 17 is one. If bit 9 is zero, prefetching of one CCW describing a data area is allowed during output-data-chaining operations.
and is not allowed during any other operations. If bit 9 is one, unlimited prefetching of CCWs, IDAWs, and associated data is allowed. It is model dependent whether prefetching is actually performed for any or all of the CCWs, IDAWs, and associated data that comprise the channel program.

**Initial-Status-Interruption Control (I)**
When the start-function bit, bit 17 of word 0, is one, bit 10 of word 0, when one, indicates that the channel subsystem is to generate an intermediate interruption condition if the subchannel becomes subchannel active (see “Initial-Status-Interruption Control (I)” on page 15-24). Bit 10 of word 0, when zero, indicates that the subchannel becoming subchannel active is not to cause an intermediate interruption condition to be generated.

The program requests the intermediate interruption condition by means of the ORB. An I/O interruption that results from that request may be due to the channel subsystem performing either a start function or a resume function. (See “Zero Condition Code (Z)” for details of the indication given by the channel subsystem when the intermediate interruption condition is cleared by TEST SUBCHANNEL.)

**Address-Limit-Checking Control (A)**
When the start-function bit, bit 17 of word 0, is one, bit 11 of word 0, when one, indicates that the channel subsystem has been requested by the program to perform address-limit checking, subject to the setting of the limit mode at the subchannel (see “Address-Limit-Checking Control (A)” on page 15-25). The address-limit-checking-control bit is meaningful only when bit 17 is one.

**Suppress-Suspended Interruption (U)**
When the start-function bit, bit 17 of word 0, is one, bit 12 of word 0, when one, indicates that the channel subsystem has been requested by the program to suppress the generation of a subchannel-suspended interruption condition when the subchannel is suspended (see “Suppress-Suspended-Interruption Control (U)” on page 15-25). When bit 12 is zero, the channel subsystem generates an intermediate interruption condition whenever the subchannel is suspended during the execution of the associated channel program. The suppress-suspended-interruption control bit is meaningful only when bit 17 is one.

**Subchannel-Control Field**
The following subchannel-control-information descriptions apply to the subchannel-control field, bits 13-31 of word 0 of the SCSW.

**Zero Condition Code (Z)**
Bit 13 of word 0, when one, indicates that the subchannel has become subchannel active and the channel subsystem has recognized an initial-status-interruption condition at the subchannel. The Z bit is meaningful only when the intermediate-status bit, bit 28 of word 0, and the start-function bit, bit 17 of word 0, are both ones.

If the initial-status-interruption-control bit, bit 10 of word 1 of the ORB, is one when START SUBCHANNEL is executed, then the subchannel becoming subchannel active causes the subchannel to be made status pending with intermediate status indicating the initial-status-interruption condition. The initial-status-interruption condition remains at the subchannel until the intermediate interruption condition is cleared by the execution of TEST SUBCHANNEL or CLEAR SUBCHANNEL. If the initial-status-interruption-control bit of the ORB is zero when START SUBCHANNEL is executed, then the subchannel becoming subchannel active does not cause an intermediate interruption condition to be generated, and the initial-status-interruption condition is not recognized.

**Extended Control (E)**
Bit 14 of word 0, when one, indicates that model-dependent information or concurrent-sense information is stored in the extended-control word (ECW). When bit 14 is zero, the contents of words 0-7 of the ECW, if stored, are unpredictable. The E bit is meaningful whenever the subchannel is status pending with alert status either alone or together with primary status, secondary status, or both.

**Programming Note:** During the execution of TEST SUBCHANNEL, the storing of words 0-7 of the ECW is a model-dependent function subject to the setting of bit 14 as described above. Therefore, the program should always provide sufficient storage to accommodate the storing of a 64-byte IRB.
Path Not Operational (N)
Bit 15 of word 0, when one, indicates that the N condition has been recognized by the channel subsystem. The N condition, in turn, indicates that one or more path-not-operational conditions have been recognized. The channel subsystem recognizes a path-not-operational condition when, during an attempted device selection in order to perform a clear, halt, resume, or start function, the device associated with the subchannel appears not operational on a channel path that is operational for the subchannel. A channel path is operational for the subchannel if the associated device appeared operational on that channel path the last time the channel subsystem attempted device selection in order to perform a clear, halt, resume, or start function. A channel path is not operational for the subchannel if the associated device appeared not operational on that channel path the last time the channel subsystem attempted device selection in order to perform a clear, halt, resume, or start function. A device appears to be operational on a channel path when the device responds to an attempted device selection.

The N bit is meaningful whenever the status-control field contains any of the indications listed below and at least one basic I/O function is also indicated at the subchannel:

- Status pending with any combination of primary, secondary, or alert status
- Status pending alone
- Status pending with intermediate status when the subchannel is also suspended

The N condition is reset whenever the execution of TEST SUBCHANNEL results in the setting of condition code 0 and the N bit is meaningful as described above.

Notes:
1. A path-not-operational condition does not imply a malfunctioning channel path. A malfunctioning channel path causes the generation of an error indication, such as interface-control check.
2. When a path-not-operational condition has been recognized and the subchannel subsequently becomes status pending with only intermediate status, the path-not-operational condition (a) continues to be recognized until the subchannel becomes status pending with primary status or becomes suspended and (b) is indicated by storing the path-not-operational bit as a one during the execution of TEST SUBCHANNEL. When a path-not-operational condition has been recognized and the channel-program execution subsequently becomes suspended, the path-not-operational condition does not remain pending if channel-program execution is subsequently resumed. Instead, the old indication is lost, and the path-not-operational indication, if any, pertains to the attempt by the channel subsystem to resume channel-program execution.

Function Control (FC)
The function-control field indicates the basic I/O functions that are indicated at the subchannel. This field may indicate the acceptance of as many as two functions. The function-control field is contained in bit positions 17-19 of the first word of the SCSW. The function-control field is meaningful at an installed subchannel whenever the subchannel is valid (see "Device Number Valid (V)" on page 15-4). The function-control field contains all zeros whenever both the activity- and status-control fields contain all zeros. The meaning of the individual bits is as follows:

**Start Function (Bit 17):** When one, bit 17 indicates that a start function has been requested and is either pending or in progress at the subchannel. A start function is requested by the execution of START SUBCHANNEL. A start function is indicated at the subchannel when condition code 0 is set during the execution of START SUBCHANNEL. The start-function indication is cleared at the subchannel when TEST SUBCHANNEL is executed and the subchannel is either status pending alone or status pending with any combination of alert, primary, or secondary status. The start-function indication is also cleared at the subchannel during the execution of CLEAR SUBCHANNEL.

**Halt Function (Bit 18):** When one, bit 18 indicates that a halt function has been requested and is either pending or in progress at the subchannel. A halt function is requested by the execution of HALT SUBCHANNEL. A halt function is indicated at the subchannel when condition code 0 is set during the execution of HALT SUBCHANNEL. The halt-function indication is cleared at the subchannel when the next status-pending condition that occurs is cleared by the execution of TEST SUBCHANNEL.
status-pending condition depends on the state of
the subchannel when HALT SUBCHANNEL is
executed. If the subchannel is subchannel active
when HALT SUBCHANNEL is executed, then the
next status-pending condition is status pending
with at least primary status indicated. If the sub-
channel is device active when HALT SUB-
CHANNEL is executed, then the next status-
pending condition is status pending with at least
secondary status indicated. If the subchannel is
suspended and status pending with intermediate
status when HALT SUBCHANNEL is executed,
then the next status-pending condition is status
pending with intermediate status. If the sub-
channel is idle when HALT SUBCHANNEL is exe-
cuted, then the next status-pending condition is
status pending alone. The halt-function indication
is also cleared at the subchannel during the exe-
cution of CLEAR SUBCHANNEL. In normal oper-
ations, this function is indicated together with bit
17; that is, there is a start function either pending
or in progress that is to be halted.

Clear Function (Bit 19): When one, bit 19 indi-
cates that a clear function has been requested
and is either pending or in progress at the sub-
channel. A clear function is requested by the exe-
cution of CLEAR SUBCHANNEL. A clear function
is indicated at the subchannel when condition
code 0 is set for CLEAR SUBCHANNEL (see
"CLEAR SUBCHANNEL" on page 14-5). The
clear-function indication is cleared at the sub-
channel when the resulting status-pending condi-
tion is cleared by TEST SUBCHANNEL.

Activity Control (AC)
The activity-control field is contained in bit posi-
tions 20-26 of the first word of the SCSW. This
field indicates the current progress of a basic I/O
function previously accepted at the subchannel.
By using the contents of this field, the program
can determine the degree of completion of the
basic I/O function. The activity-control field is
meaningful at an installed subchannel whenever
the subchannel is valid (see "Device Number Valid
(V)" on page 15-4). However, if an IFCC or CCC
condition is detected during the performance of a
basic I/O function and that function is indicated as
pending, I/O operations may or may not have been
performed at the device. The activity-control
bits are defined as follows:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Resume pending</td>
</tr>
<tr>
<td>21</td>
<td>Start pending</td>
</tr>
<tr>
<td>22</td>
<td>Halt pending</td>
</tr>
<tr>
<td>23</td>
<td>Clear pending</td>
</tr>
<tr>
<td>24</td>
<td>Subchannel active</td>
</tr>
<tr>
<td>25</td>
<td>Device active</td>
</tr>
<tr>
<td>26</td>
<td>Suspended</td>
</tr>
</tbody>
</table>

When an SCSW is stored that has the status-
pending bit of the status-control field zero and all
zeros in the activity-control field, the subchannel is
said to be idle or in the idle state.

Note: All conditions that are represented by the
bits in the function-control field and by the
resume-pending, start-pending, halt-pending,
clear-pending, subchannel-active, and suspended
bits in the activity-control field are reset at the sub-
channel when TEST SUBCHANNEL is executed
and the subchannel is (1) status pending alone,
(2) status pending with primary status, (3) status
pending with alert status, or (4) status pending
with intermediate status and is also suspended.

Resume-Pending (Bit 20): When one, bit 20
indicates that the subchannel is resume pending.
The channel subsystem may or may not be in the
process of performing the start function. The sub-
channel becomes resume pending when condition
code 0 is set for RESUME SUBCHANNEL. The
point at which the subchannel is no longer resume
pending is a function of the subchannel state
existing when the resume-pending condition is
recognized and the state of the device if channel-
program execution is resumed.

If the subchannel is in the suspended state when
the resume-pending condition is recognized, the
CCW that caused the suspension is refetched,
the setting of the suspend flag is examined, and one
of the following actions is taken by the channel
subsystem:

1. If the CCW suspend flag is one, the device is
   not selected, the subchannel is no longer
   resume pending, and the channel-program
   execution remains suspended.
2. If the CCW suspend flag is zero, the channel
   subsystem attempts to resume channel-
   program execution by performing a modified
   start function. The resumption of channel-
   program execution appears to the device as
   the initiation of a new channel-program exe-
   cution. The resume function causes the
   channel subsystem to perform the path-
   management operation as if a new start func-
tion were being initiated, using the ORB parameters previously passed to the subchannel by START SUBCHANNEL with the exception that the channel-program address is the address of the CCW that caused the suspension of the channel-program execution.

The subchannel remains resume pending when, during the performance of the start function, the channel subsystem (1) determines that it is not possible to attempt to initiate the I/O operation for the first command, (2) determines that an attempt to initiate the I/O operation for the first command does not result in the command being accepted, or (3) detects an IFCC or CCC condition and is unable to determine whether the first command has been accepted. (See "Start Function and Resume Function" on page 15-18.)

The subchannel is no longer resume pending when any of the following events occurs:

- a. While performing the start function, the subchannel becomes subchannel-and-device active or device active only, or the first command is accepted with channel-end and device-end initial status and the CCW does not specify command chaining.
- b. CLEAR SUBCHANNEL is executed.
- c. TEST SUBCHANNEL clears any combination of primary, secondary, and alert status or clears the status-pending condition alone.
- d. TEST SUBCHANNEL clears intermediate status while the subchannel is suspended.
- e. CANCEL SUBCHANNEL is executed with a resulting condition code 0.

If the subchannel is not in the suspended state when the resume-pending condition is recognized, the CCW suspend flag of the most recently fetched CCW, if any, is examined, and one of the following actions is taken by the channel subsystem:

1. If a CCW has not been fetched or the suspend flag of the most recently fetched CCW is zero, the subchannel is no longer resume pending, and the resume function is not performed.

2. If the suspend flag of the most recently fetched CCW is one, the subchannel is no longer resume pending, and the CCW is refetched. The subchannel proceeds with channel-program execution if the suspend flag of the refetched CCW is zero. The subchannel suspends channel-program execution if the suspend flag of the refetched CCW is one.

Some models recognize a resume-pending condition only after a CCW having an S flag validly set to one is fetched. Therefore, if a subchannel is resume pending and, during the execution of the channel program, no CCW is fetched that has an S flag validly set to one, the subchannel remains resume pending until the primary interruption condition is cleared by TEST SUBCHANNEL.

Start-Pending (Bit 21): When one, bit 21 indicates that the subchannel is start pending. The channel subsystem may or may not be in the process of performing the start function. The subchannel becomes start pending when condition code 0 is set for START SUBCHANNEL. The subchannel remains start pending when, during the performance of the start function, the channel subsystem (1) determines that it is not possible to attempt to initiate the I/O operation for the first command, (2) determines that an attempt to initiate the I/O operation for the first command does not result in the command being accepted, or (3) detects an IFCC or CCC condition and is unable to determine whether the first command has been accepted. (See "Start Function and Resume Function" on page 15-18.)

The subchannel becomes no longer start pending when any of the following occurs:

1. While performing the start function, the subchannel becomes subchannel-and-device active or device active only, or the first command is accepted with channel-end and device-end initial status and the CCW does not specify command chaining.

2. The subchannel becomes suspended because of a suspend flag validly set to one in the first CCW.

3. CLEAR SUBCHANNEL is executed.

4. TEST SUBCHANNEL clears any combination of primary, secondary, and alert status or clears the status-pending condition alone.
5. CANCEL SUBCHANNEL is executed with a resulting condition code 0.

**Halt-Pending (Bit 22):** When one, bit 22 indicates that the subchannel is halt pending. The channel subsystem may or may not be in the process of performing the halt function. The subchannel becomes halt pending when condition code 0 is set for HALT SUBCHANNEL. The subchannel remains halt pending when, during the performance of the halt function, the channel subsystem (1) determines that it is not possible to attempt to issue the halt signal to the device, (2) determines that the attempt to issue the halt signal to the device is not successful, or (3) detects an IFCC or CCC condition and is unable to determine whether the halt signal is issued to the device. (See "Halt Function" on page 15-15.)

The subchannel is no longer halt pending when any of the following occurs:

1. While performing the halt function, the channel subsystem determines that the halt signal has been issued to the device.
2. CLEAR SUBCHANNEL is executed.
3. TEST SUBCHANNEL clears any combination of primary, secondary, and alert status or clears the status-pending condition alone.
4. TEST SUBCHANNEL clears intermediate status while the subchannel is suspended.

**Clear-Pending (Bit 23):** When one, bit 23 indicates that the subchannel is clear pending. The channel subsystem may or may not be in the process of performing the clear function. The subchannel becomes clear pending when condition code 0 is set for CLEAR SUBCHANNEL. The subchannel remains clear pending when, during performance of the clear function, the channel subsystem (1) determines that it is not possible to attempt to issue the clear signal to the device, (2) determines that the attempt to issue the clear signal to the device is not successful, or (3) detects an IFCC or CCC condition and is unable to determine whether the clear signal is issued to the device. (See "Clear Function" on page 15-14.)

The subchannel is no longer clear pending when either of the following occurs:

1. While performing the clear function, the channel subsystem determines that the clear signal has been issued to the device.
2. TEST SUBCHANNEL clears the status-pending condition alone.

**Subchannel Active (Bit 24):** When one, bit 24 indicates that the subchannel is subchannel active. A subchannel is said to be subchannel active when an I/O operation is currently being performed at the subchannel. The subchannel becomes subchannel active when the first command is accepted and the start function or resume function is not immediately concluded at the subchannel. (See "Immediate Conclusion of I/O Operations" on page 15-43.) The subchannel is no longer subchannel active when any of the following occurs:

1. The subchannel becomes suspended.
2. The subchannel becomes status pending with primary status.
3. CLEAR SUBCHANNEL is executed.
4. The device appears not operational during performance of a halt function.

The subchannel does not become subchannel active during performance of the function specified by either a HALT SUBCHANNEL or a CLEAR SUBCHANNEL instruction.

**Device Active (Bit 25):** When one, bit 25 indicates that the subchannel is device active. A subchannel is said to be device active when an I/O operation is currently in progress at the associated device. The subchannel becomes device active when the first command is accepted. The subchannel is no longer device active when any of the following occurs:

1. The subchannel becomes suspended.
2. The subchannel becomes status pending with secondary status.
3. CLEAR SUBCHANNEL is executed.
4. The device appears not operational during performance of a halt function.

If the subchannel is not start pending or if the status accepted from the device also describes an alert condition, the subchannel becomes status pending with secondary status. After the status has been accepted from the device, the device is capable of accepting a command for performing a
new I/O operation. If the subchannel is start pending and the status is device end or device end with control unit end, then the channel subsystem discards the status and performs the start function for the new channel program. (See “Start Function and Resume Function” on page 15-18) In this situation, the subchannel does not become status pending with the secondary interruption condition, and the status is not made available to the program.

The subchannel does not become device active during performance of the functions specified by either a HALT SUBCHANNEL or a CLEAR SUBCHANNEL instruction.

**Suspected (Bit 26):** When one, bit 26 indicates that the subchannel is suspected. A subchannel is said to be suspected when channel-program execution is currently suspended. The subchannel becomes suspected as part of the suspend function. (See “Suspension of Channel-Program Execution” on page 15-38)

The subchannel is no longer suspected when any of the following occurs:

1. As part of the resume function following the execution of RESUME SUBCHANNEL when the subchannel becomes subchannel-and-device active or device active only, or the first command is accepted for channel-end and device-end initial status, with or without status modifier, and the CCW does not specify command chaining.
2. CLEAR SUBCHANNEL is executed.
3. TEST SUBCHANNEL clears any combination of primary, secondary, and alert status or clears the status-pending condition alone.
4. TEST SUBCHANNEL clears intermediate status while the halt function is specified.
5. CANCEL SUBCHANNEL is executed with a resulting condition code 0.

**Programming Note:** When an SCSW is stored by STORE SUBCHANNEL or TEST SUBCHANNEL following CLEAR SUBCHANNEL but prior to the subchannel becoming status pending, and the subchannel-active bit, bit 24 of word 0, is stored as zero, this does not mean that data transfer has stopped for the device. The program cannot determine whether data transfer has stopped until the subchannel becomes status pending as a result of performing the clear function.

**Status Control (SC)**

The status-control field is contained in bit positions 27-31 of the first word of the SCSW. This field provides the program with a summary-level indication of the interruption condition described by either subchannel or device status, the Z bit, or, in the case of the subchannel-suspended interruption, the suspended bit, bit 26. More than one summary indication may be signaled as a result of existing conditions at the subchannel. Whenever the subchannel is enabled (see “Enabled (E)” on page 15-2) and at least bit 31 is one, the subchannel is said to be status pending. Whenever the subchannel is disabled, the subchannel is not made status pending. Bit 31 of SCSW word 0 is meaningful at an installed subchannel whenever the subchannel is valid (see “Device Number Valid (V)” on page 15-4); bits 27-30 are meaningful when bit 31 is one. The status-control bits are defined as follows:

**Alert Status (Bit 27):** When one (and when the status-pending bit is also one), bit 27 indicates an alert interruption condition exists. In such a case, the subchannel is said to be status pending with alert status. An alert interruption condition is recognized when alert status is present at the subchannel. Alert status may be subchannel status or device status. Alert status is status generated by either the channel subsystem or the device under any of the following conditions:

- The subchannel is idle (activity-control bits 20-26 and status-control bit 31 are zeros).
- The subchannel is start pending, and the status condition precludes initiation of the I/O operation.
- The subchannel is subchannel-and-device active, and the status condition has suppressed command chaining or would have suppressed command chaining if chaining had been specified (see “Chaining” on page 15-31).
- The subchannel is subchannel-and-device active, command chaining is not specified, the execution of the channel program has just been concluded, and the status presented by the device is attempting to alter the sequential execution of commands (see the publication ESA/390 Common I/O-Device Commands,
The subchannel is device active only, and the status presented by the device is other than device end, control unit end, or device end and control unit end.

- The subchannel is suspended (bit 26 is one).

If the subchannel is start pending when an alert interruption condition is recognized, the subchannel becomes status pending with alert status, deferred condition code 1 is set, the start-pending bit remains one, and the performance of the pending I/O operation is not initiated.

When TEST SUBCHANNEL is executed and stores an SCSW with the alert-status bit and the status-pending bit as ones in the IRB, the alert interruption condition is cleared at the subchannel. The alert interruption condition is also cleared during the execution of CLEAR SUBCHANNEL.

Whenever alert status is present at the subchannel, it is brought to the attention of the program. Examples of alert status include attention, device end (which signals a transition from the not-ready to the ready state), incorrect length, program check, and unit check.

**Intermediate Status (Bit 28):** When one (and when the status-pending bit is also one), bit 28 indicates an intermediate interruption condition exists. In such a case, the subchannel is said to be status pending with intermediate status. Intermediate status can be indicated when the Z bit (of the subchannel-control field), the suspended bit (of the activity-control field), or the PCI bit (of the subchannel-status field) is one.

When the initial-status-interruption-control bit is one in the ORB, the subchannel becomes status pending with intermediate status (the Z bit indicated) only after the subchannel is subchannel active. If the subchannel does not become subchannel active, the Z condition is not generated.

When suspend control is specified and the generation of an intermediate interruption condition due to suspension is not suppressed in the ORB, then the subchannel can become status pending with intermediate status due to suspension if a CCW becomes current that contains the suspend flag set to one. When the suspend flag is specified in the first CCW of a channel program, channel-program execution is suspended, and the subchannel becomes status pending with intermediate status (the suspended bit indicated) before the command in the first CCW is transferred to the device. When the suspend flag is specified in a CCW fetched during command chaining, then channel-program execution is suspended, and the subchannel becomes status pending with intermediate status (the suspended bit is indicated), only after the execution of the preceding CCW is complete.

When the PCI flag is specified in a CCW, the generation of an intermediate interruption condition due to PCI depends on whether the CCW is the first CCW of the channel program. When the PCI flag is specified in the first CCW of a channel program, the subchannel becomes status pending with intermediate status (the PCI bit indicated) only after initial status is received for the first CCW of the channel program indicating the command has been accepted. When the PCI flag is specified in a CCW fetched while chaining, the subchannel becomes status pending with intermediate status (the PCI bit indicated) only after the execution of the preceding CCW is complete. If chaining occurs before an interruption condition containing PCI is cleared by TEST SUBCHANNEL, the condition is carried over to the next CCW. This carry-over occurs during both data and command chaining, and, in either case, the condition is propagated through the transfer-in-channel command.

If the subchannel is status pending with intermediate status when HALT SUBCHANNEL is executed, the intermediate interruption condition remains at the subchannel, but the interruption request, if any, is withdrawn, and the subchannel becomes no longer status-pending. The subchannel remains no longer status pending until performance of the halt function has ended. The subchannel then becomes status pending with intermediate status indicated (possibly together with any combination of primary, secondary, and alert status).

When TEST SUBCHANNEL is executed and stores an SCSW with the intermediate-status bit and the status-pending bit as ones in the IRB, the intermediate interruption condition is cleared at the subchannel. The intermediate interruption condi-
tion is also cleared at the subchannel during the execution of CLEAR SUBCHANNEL.

**Primary Status (Bit 29):** When one (and when the status-pending bit is also one), bit 29 indicates a primary interruption condition exists. In such a case, the subchannel is said to be status pending with primary status. A primary interruption condition is a solicited interruption condition that indicates the completion of the start function at the subchannel. The primary interruption condition is described by the SCSW stored. When an I/O operation is terminated by HALT SUBCHANNEL but the halt signal is not issued to the device because the device appeared not operational, the subchannel is made status pending with primary status (and secondary status) with both the subchannel-status field and the device-status field set to zero.

When TEST SUBCHANNEL is executed and stores an SCSW with the primary-status bit and the status-pending bit as ones in the IRB, the primary interruption condition is cleared at the subchannel. The primary interruption condition is also cleared at the subchannel during the execution of CLEAR SUBCHANNEL.

**Secondary Status (Bit 30):** When one (and when the status-pending bit is also one), bit 30 indicates a secondary interruption condition exists. In such a case, the subchannel is said to be status pending with secondary status. A secondary interruption condition is a solicited interruption condition that normally indicates the completion of the I/O operation at the device. The secondary interruption condition is described by the SCSW stored.

When an I/O operation is terminated by HALT SUBCHANNEL but the halt signal is not issued to the device because the device appeared not operational, the subchannel is made status pending with secondary status (and primary status if the subchannel is also subchannel active) with zeros for subchannel and device status.

When TEST SUBCHANNEL is executed and stores an SCSW with the secondary-status bit as one in the IRB, the secondary interruption condition is cleared at the subchannel. The secondary interruption condition is also cleared at the subchannel during the execution of CLEAR SUBCHANNEL.

**Status-Pending (Bit 31):** When one, bit 31 indicates that the subchannel is status pending and that information describing the cause of the interruption condition is available to the program. The subchannel becomes status pending whenever intermediate, primary, secondary, or alert status is generated. When HALT SUBCHANNEL is executed, designating a subchannel that is idle, the subchannel becomes status pending subsequent to performance of the halt function to notify the program that the halt function has been completed. When TEST SUBCHANNEL is executed, thus storing an SCSW with the status-pending bit as one in the IRB, the status-pending condition is cleared at the subchannel. The status-pending condition is also cleared at the subchannel during the execution of CLEAR SUBCHANNEL. When CLEAR SUBCHANNEL is executed and the designated subchannel is operational, the subchannel becomes status pending subsequent to performance of the clear function to notify the program that the clear function has been completed.

**Note:** The status-pending bit, in conjunction with the remaining bits of the status-control field, indicates the type of status condition. For example, if bits 29 and 31 are ones, the subchannel is status pending with primary status. Alternatively, if only bit 31 is one, then the subchannel is said to be status pending or status pending alone. If only bit 31 is one in the status-control field, the settings of all bits in the subchannel-status and device-status fields are unpredictable. If bit 31 is not one, then the remaining bits of the status-control field are not meaningful.

**CCW-Address Field**

Bits 1-31 of word 1 form an absolute address. The address indicated is a function of the subchannel state when the SCSW is stored, as indicated in Figure 16-4 on page 16-19. When the subchannel-status field indicates channel-control check, channel-data check, or interface-control check, the CCW-address field is usable for recovery purposes if the CCW-address field-validity flag in the ESW is one.

**Programming Note:** When a CCW address, either detected in the channel-program address (see "Channel-Program Address" on page 15-26) or generated during chaining, would cause the channel subsystem to fetch a CCW from a
location greater than \(2^{24} - 1\) while format-0 CCWs are specified for the operation, the invalid address is stored in the CCW-address field of the SCSW without truncation. If the invalid address causes the channel subsystem, while chaining, to fetch a CCW from a location greater than \(2^{31} - 1\) while format-1 CCWs are specified for the operation, the rightmost 31 bits of the invalid address are stored in the CCW-address field.

<table>
<thead>
<tr>
<th>Subchannel State(^1)</th>
<th>CCW Address(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start pending (UUUUU/AIPSX)(^3)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Start pending and device active (UUUU0/AIPSX)(^3)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Subchannel-and-device active (UUUU0/AIPSX)(^3)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Device active only (UUUU0/AIPSX)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Suspended (YYYYY/AIPSX)(^3)</td>
<td>See note 1</td>
</tr>
<tr>
<td>Status pending (10001/AIPSX) because of unsolicited alert status from the device while the subchannel was start pending(^3)</td>
<td>Channel-program address + 8</td>
</tr>
<tr>
<td>Status pending (0Y111/AIPSX) because the device appeared not operational on all paths(^3)</td>
<td>Channel-program address + 8</td>
</tr>
<tr>
<td>Status pending (10011/AIPSX) because of solicited alert status from the device while the subchannel was start pending and device active(^3)</td>
<td>Channel-program address + 8</td>
</tr>
<tr>
<td>Status pending (10111/AIPSX) because of solicited alert status generated by the channel subsystem while the subchannel was start pending(^3) or start pending and device active(^3)</td>
<td>See note 2</td>
</tr>
<tr>
<td>Status pending (010001/AIPSX) for the program-controlled-interruption condition while the subchannel was subchannel-and-device active(^3)</td>
<td>CCW + 8 of the CCW that contained the last recognized PCI, or 8 higher than a CCW that has subsequently become current</td>
</tr>
<tr>
<td>Status pending (010001/AIPSX) for the initial-status-interruption condition while the subchannel was subchannel-and-device active(^3)</td>
<td>CCW + 8 of the CCW causing the intermediate interruption condition, or a CCW that has subsequently become current</td>
</tr>
<tr>
<td>Status pending (1Y1Y1/AIPSX); termination occurred because of program check caused by one of the following conditions:(^3)</td>
<td>Channel-program address + 8</td>
</tr>
<tr>
<td>Bit 24 of word 1 of the ORB set to one; incorrect-length-indication-suppression facility not installed</td>
<td>Channel-program address + 8</td>
</tr>
<tr>
<td>Unused bits in ORB not set to zeros</td>
<td>Channel-program address + 8</td>
</tr>
<tr>
<td>Invalid CCW-address specification in transfer in channel (TIC)</td>
<td>Address of TIC + 8</td>
</tr>
<tr>
<td>Invalid CCW-address specification in the channel-program address in the ORB</td>
<td>Channel-program address + 8(^4)</td>
</tr>
</tbody>
</table>

Figure 16-4 (Part 1 of 4). CCW Address as Function of Subchannel State
<table>
<thead>
<tr>
<th>Subchannel State</th>
<th>CCW Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid CCW address in TIC</td>
<td>Address of TIC + 8</td>
</tr>
<tr>
<td>Invalid CCW address in the channel-program address in the ORB</td>
<td>Channel-program address + 8^6</td>
</tr>
<tr>
<td>Invalid CCW address while chaining</td>
<td>Invalid CCW address + 8</td>
</tr>
<tr>
<td>Invalid command code</td>
<td>Address of invalid CCW + 8^6</td>
</tr>
<tr>
<td>Invalid count</td>
<td>Address of invalid CCW + 8^6</td>
</tr>
<tr>
<td>Invalid IDAW-address specification</td>
<td>Address of invalid CCW + 8^6</td>
</tr>
<tr>
<td>Invalid IDAW address in a CCW</td>
<td>Address of invalid CCW + 8^6</td>
</tr>
<tr>
<td>Invalid IDAW address while sequentially fetching IDAWs</td>
<td>Address of current CCW + 8</td>
</tr>
<tr>
<td>Invalid data-address specification, format 1</td>
<td>Address of invalid CCW + 8^6</td>
</tr>
<tr>
<td>Invalid data address in a CCW</td>
<td>Address of invalid CCW + 8^6</td>
</tr>
<tr>
<td>Invalid data address while sequentially accessing storage</td>
<td>Address of current CCW + 8</td>
</tr>
<tr>
<td>Invalid data address in IDAW</td>
<td>Address of current CCW + 8</td>
</tr>
<tr>
<td>Invalid IDAW specification</td>
<td>Address of current CCW + 8</td>
</tr>
<tr>
<td>Invalid CCW, format 0 or 1, for a CCW other than a TIC</td>
<td>Address of invalid CCW + 8^6</td>
</tr>
<tr>
<td>Invalid suspend flag — CCW fetched during data chaining has suspend flag set to one</td>
<td>Address of invalid CCW + 8</td>
</tr>
<tr>
<td>Invalid suspend flag — CCW has suspend flag set to one, but suspend control was not specified in the ORB</td>
<td>Address of invalid CCW + 8</td>
</tr>
<tr>
<td>Invalid CCW, format 1, for a TIC</td>
<td>Address of TIC + 8</td>
</tr>
<tr>
<td>Invalid sequence — two TICs</td>
<td>Address of second TIC + 8</td>
</tr>
<tr>
<td>Invalid sequence — 256 or more CCWs without data transfer</td>
<td>Address of 256th CCW + 8</td>
</tr>
<tr>
<td>Status pending (Y1Y1/AIPSX); termination occurred because of protection check detected as follows:*</td>
<td>Address of the protected CCW + 8^6</td>
</tr>
<tr>
<td>On a CCW access</td>
<td>Address of current CCW + 8</td>
</tr>
<tr>
<td>On data or an IDAW access</td>
<td>Address of current CCW + 8</td>
</tr>
</tbody>
</table>

Figure 16-4 (Part 2 of 4). CCW Address as Function of Subchannel State
<table>
<thead>
<tr>
<th>Subchannel State</th>
<th>CCW Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status pending (1Y1Y1/AIPSX); termination occurred because of chaining check(^3)</td>
<td>Address of current CCW + 8</td>
</tr>
<tr>
<td>Status pending (YY1Y1/AIPSX); termination occurred under count control(^3)</td>
<td>Address of current CCW + 8(^6)</td>
</tr>
<tr>
<td>Status pending (1Y1Y1/AIPSX); operation prematurely terminated by the device because of alert status(^3)</td>
<td>Address of current CCW + 8(^6)</td>
</tr>
<tr>
<td>Status pending (YYYY1/AIPSX) after termination by HALT SUBCHANNEL and the activity-control-field bits indicated below set to ones:</td>
<td></td>
</tr>
<tr>
<td>Status pending alone</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Start pending(^3)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Device active and start pending(^3)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Device active</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Subchannel active and device active(^3)</td>
<td>CCW + 8 of the last-executed CCW</td>
</tr>
<tr>
<td>Suspended</td>
<td>CCW + 8 of CCW causing suspension</td>
</tr>
<tr>
<td>Suspended and resume pending</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Status pending (00001/AIPSX) after termination by CLEAR SUBCHANNEL</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Status pending (YY1Y1/AIPSX); operation completed normally at the subchannel(^3)</td>
<td>CCW + 8 of the last-executed CCW(^6)</td>
</tr>
<tr>
<td>Status pending (00011/AIPSX)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Status pending (10001/AIPSX)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Status pending (00001/AIPSX)</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Status pending (11111/AIPSX); command chaining suppressed because of alert status other than channel-control check or interface-control check(^3)</td>
<td>Address of current CCW + 8(^6)</td>
</tr>
<tr>
<td>Status pending (1YYYY1/AIPSX) because of alert status for channel-control check or interface-control check(^3)</td>
<td>See note 3(^6)</td>
</tr>
<tr>
<td>Status pending (1Y1Y1/AIPSX) because of channel-data check(^3)</td>
<td>Address of current CCW + 8(^6)</td>
</tr>
</tbody>
</table>

*Figure 16-4 (Part 3 of 4). CCW Address as Function of Subchannel State*
Explanation:

1. The meaning of the notation used in this column is as follows:
   - A Alert status
   - I Intermediate status
   - P Primary status
   - S Secondary status
   - X Status pending
   The possible combination of status-control-bit settings is shown to the left of the "/" symbol by the use of these symbols:
   - O Corresponding condition is not indicated.
   - I Corresponding condition is indicated.
   - U Unpredictable. The corresponding condition is not meaningful when the subchannel is not status pending.
   - Y The corresponding condition is not significant and is indicated as a function of the subchannel state.

2. A CCW becomes current when (1) it is the first CCW of a channel program and has been fetched, (2) while command chaining, the previous CCW is no longer current and the new CCW has been fetched, or (3) in the case of data chaining, the new CCW takes over control of the I/O operation (see the section “Data Chaining” in Chapter 15, “Basic I/O Functions”). If chaining is not specified or is suppressed, a CCW is no longer current and becomes the last-executed CCW when secondary status has been accepted by the channel subsystem. During command chaining, a CCW is no longer current when device-end status has been accepted or, in the case of data chaining, when the last byte of data for that CCW has been accepted.

3. The subchannel may also be resume pending.

4. The stored address is the channel-program address (in the ORB) + 8 even though it is either invalid or protected.

5. The stored address is the address of the current CCW + 8 even though it is either invalid or protected.

6. Incorrect length is indicated as a function of the setting of the suppress-length-indication flag in the current CCW (see the section “Channel-Command Word” in Chapter 15, “Basic I/O Functions”).

Notes:

1. Unless the subchannel is also resume pending, the address stored is the address of the CCW that caused suspension, plus 8. Otherwise, the address stored is unpredictable.

2. The address of the CCW is given as a function of the alert status indicated. For example, if a program-check or protection-check condition is recognized, the CCW address stored is the same as for the entry for program check or protection check, respectively, in this table. Alternatively, if alert status for interface-control check or channel-control check is indicated, the CCW address stored is either the channel-program address (in the ORB) + 8 or invalid as specified by the field-validity flags in the subchannel logout.

3. Bit 21 of the subchannel-logout information, when stored as one, indicates that the address is CCW + 8 of the last-fetched CCW if the command for the CCW has not been accepted by the device. If the command has been accepted by the device at the time the error condition is recognized, then the address stored is the address of the CCW + 8 of the last-executed CCW.

Figure 16-4 (Part 4 of 4). CCW Address as Function of Subchannel State
Device-Status Field

Device-status conditions are generated by the I/O device and are presented to the channel subsystem over the channel path. The timing and causes of these conditions for each type of device are specified in the System Library publication for the device. The device-status field is meaningful whenever the subchannel is status pending with any combination of primary, secondary, intermediate, or alert status. Whenever the subchannel is status pending with intermediate status alone, the device-status field is zero. When the subchannel-status field indicates channel-control check, channel-data check, or interface-control check, the device-status field is usable for recovery purposes if the device-status field-validity flag in the ESW is one. When the subchannel is status pending with deferred condition code 3 indicated, the contents of the device-status field are not meaningful.

If, within a system, the I/O device is accessible from more than one channel path, status related to channel-subsystem-initiated operations in the single-path mode (solicited status) is signaled over the initiating channel path. Devices operating in the multipath mode may signal solicited status over any channel path that belongs to the same path group as the initiating channel path. The handling of conditions not associated with I/O operations (unsolicited alert status), such as attention, unit exception, and device end due to transition from the not-ready to the ready state, depends on the type of device and condition and is specified in the System Library publication for the device.

The channel subsystem does not modify the status bits received from the I/O device. These bits appear in the SCSW as received over the channel path. For more information on the status bits received from the I/O device, see the publication ESA/390 Common I/O-Device Commands, SA22-7204.

Subchannel-Status Field

Subchannel-status conditions are detected and indicated in the SCSW by the channel subsystem. Except for the conditions caused by equipment malfunctioning, they can occur only while the channel subsystem is involved with the performance of a halt, resume, or start function. The subchannel-status field is meaningful whenever the subchannel is status pending with any combination of primary, secondary, intermediate, or alert status. Individual bits contained in the subchannel-status field may be unpredictable even when the subchannel-status field is meaningful. When the subchannel is status pending with deferred condition code 3 indicated, the contents of the subchannel-status field are not meaningful.

Program-Controlled Interruption

An intermediate interruption condition is generated after a CCW with the program-controlled-interruption (PCI) flag set to one becomes the current CCW. The I/O interruption due to the PCI flag may be delayed an unpredictable amount of time because of masking of the interruption request or other activity in the system. Detection of the PCI condition does not affect the progress of the I/O operation.

Incorrect Length

Incorrect length occurs when the number of bytes contained in the storage areas assigned for the I/O operation is not equal to the number of bytes requested or offered by the I/O device. Incorrect length is indicated for one of the following reasons:

Long Block on Input: During a read, read-backward, or sense operation, the device attempted to transfer one or more bytes to main storage after the assigned main-storage areas were filled, or the device indicated that more data could have been transferred if the count had been larger. The extra bytes have not been placed in main storage. The count in the SCSW is zero.

Long Block on Output: During a write or control operation, the device requested one or more bytes from the channel subsystem after the assigned main-storage areas were exhausted, or the device indicated that more data could have been transferred if the count had been larger. The count in the SCSW is zero.
Short Block on Input: The number of bytes transferred during a read, read-backward, or sense operation is insufficient to fill the main-storage areas assigned to the operation. The count in the SCSW is not zero.

Short Block on Output: The device terminated a write or control operation before all information contained in the assigned main-storage areas was transferred to the device. The count in the SCSW is not zero.

The incorrect-length indication is suppressed when the current CCW has the SLI flag set to one and the CD flag set to zero. The indication does not occur for operations rejected during the initiation sequence. The indication also does not occur for immediate operations when the count field is nonzero and the subchannel is in the incorrect-length-suppression mode. The incorrect-length indication is not meaningful when the count field of the SCSW is not meaningful.

Presence of the incorrect-length condition suppresses command chaining unless the SLI flag in the CCW is one or unless the condition occurs in an immediate operation when the subchannel is in the incorrect-length-suppression mode.

Program Check
Program check occurs when programming errors are detected by the channel subsystem. The condition can be due to the following causes:

Invalid CCW-Address Specification: The channel-program address (CPA) or the transfer-in-channel command does not designate the CCW on a doubleword boundary, or bit 0 of the CPA or bit 32 of a format-1 CCW specifying the transfer-in-channel command is not zero.

Invalid CCW Address: The channel subsystem has attempted to fetch a CCW from a main-storage location that is not available. An invalid CCW address can occur because the program has designated an invalid address in the channel-program-address field of the ORB or in the transfer-in-channel command or because, on chaining, the channel subsystem attempts to fetch a CCW from an unavailable location. A main-storage location is unavailable when any of the following conditions is detected:

1. The absolute CCW address does not correspond to a physical location.
2. Format-0 CCWs are specified in the ORB, and the absolute CCW address is greater than $2^{24} - 1$.
3. Format-1 CCWs are specified in the ORB, and the absolute CCW address is greater than $2^{31} - 1$.

Invalid Command Code: There are zeros in the four rightmost bit positions of the command code in the CCW designated by the CPA or in a CCW fetched on command chaining. The command code is not tested for validity during data chaining.

Invalid Count, Format 0: A CCW, which is other than a CCW specifying transfer in channel, contains zeros in bit positions 48-63.

Invalid Count, Format 1: A CCW that specifies data chaining or a CCW fetched while data chaining contains zeros in bit positions 16-31.

Invalid IDAW-Address Specification: Indirect data addressing is specified, and either of the following conditions is detected:

1. The ORB specifies format-1 IDAWs, and the contents of the data-address field in the CCW do not designate the first IDAW on a word boundary; that is, bits 30 and 31 (format-0 CCW) or 62 and 63 (format-1 CCW) are not zeros.
2. The ORB specifies format-2 IDAWs, and the contents of the data-address field in the CCW do not designate the first IDAW on a doubleword boundary; that is, bits 29-31 (format-0 CCW) or 61-63 (format-1 CCW) are not zeros.

Invalid IDAW Address: The channel subsystem has attempted to fetch an IDAW from a main-storage location that is not available. An invalid IDAW address can occur because the program has designated an invalid address in a CCW or in the data-address field in the CCW that specifies indirect data addressing or because the channel subsystem, on sequentially fetching IDAWs, attempts to fetch from an unavailable location. A main-storage location is unavailable when any of the following conditions is detected:

1. The absolute IDAW address does not correspond to a physical location.
2. Format-0 CCWs are specified in the ORB, and the absolute IDAW address is greater than $2^{24} - 1$.

3. Format-1 CCWs are specified in the ORB, and the absolute IDAW address is greater than $2^{31} - 1$.

Invalid Data-Address Specification: Bit 32 of a format-1 CCW is not zero.

Invalid Data Address: When any of the following conditions is detected, an invalid data address is recognized by the channel subsystem.

1. Use of the data address has caused the channel subsystem to attempt to wrap from the maximum storage address to zero.
2. Use of the data address has caused the channel subsystem to attempt to wrap from zero to the maximum storage address during a read-backward operation.
3. The channel subsystem has attempted to transfer data to a storage location that is unavailable.

An invalid data address can occur because the program has designated an unavailable location in a CCW or in an IDAW, or because the channel subsystem, on sequentially accessing storage, attempted to access an unavailable location. A main-storage location is unavailable when any of the following conditions is detected:

1. The absolute address of the location does not correspond to a physical location.
2. Format-0 CCWs are specified in the ORB, indirect data addressing is not specified in the CCW, and the absolute address is greater than $2^{24} - 1$.
3. Format-1 CCWs are specified in the ORB, indirect data addressing is not specified in the CCW, and the absolute address is greater than $2^{31} - 1$.
4. Format-1 IDAWs are specified in the ORB, indirect data addressing is specified in the CCW, and the absolute address is greater than $2^{31} - 1$.
5. The absolute address is outside the addressing range specified by SET ADDRESS LIMIT, and the limit mode at the subchannel is active.

Note: The maximum storage address is determined as a function of the CCW and IDAW formats used. When an IDAW is not used, the maximum storage address is a function of the CCW format specified, as follows:

1. When 24-bit (format 0) CCWs are specified, the maximum storage address recognized by the channel subsystem is $2^{24} - 1$.
2. When 31-bit (format 1) CCWs are specified, the maximum storage address recognized by the channel subsystem is $2^{31} - 1$.

When an IDAW is used, the maximum storage address is a function of the IDAW format specified, as follows:

1. When 31-bit (format 1) IDAWs are specified, the maximum storage address recognized by the channel subsystem is $2^{31} - 1$.
2. When 64-bit (format 2) IDAWs are specified, the maximum storage address recognized by the channel subsystem is $2^{64} - 1$.

Invalid IDAW Specification: When any of the following conditions is detected, an invalid IDAW specification is recognized by the channel subsystem:

1. Bit 0 of a format-1 IDAW is not zero.
2. A second or subsequent format-1 IDAW does not designate the location of the beginning byte of a 2K-byte block or, for read-backward operations, the location of the ending byte of a 2K-byte block.
3. A second or subsequent format-2 IDAW does not designate the location of the beginning byte of a 2K-byte or 4K-byte block, as required by the 2K-IDAW control in the ORB, or, for read-backward operations, the location of the ending byte of a 2K-byte or 4K-byte block.

Invalid CCW, Format 0: A CCW other than a CCW specifying transfer in channel does not contain a zero in bit position 39.

Invalid CCW, Format 1: A CCW other than a CCW specifying transfer in channel does not contain a zero in bit position 15, or a CCW specifying transfer in channel does not contain zeros in bit positions 0-3 and 8-31.
Invalid Suspend Flag: A format-0 or format-1 CCW fetched during data chaining, other than a CCW specifying transfer in channel, does not contain a zero in bit position 38 or 14, respectively. A CCW other than a CCW specifying transfer in channel does not contain a zero in bit position 38 for a format-0 CCW or bit position 14 for a format-1 CCW, and suspend control was not specified by bit 4 of word 1 of the ORB.

Invalid ORB Format: One or more reserved bit positions in the operation-request block (ORB) is not zero. (See "Operation-Request Block" on page 15-22 for more information.) If the incorrect-length-indication-suppression facility is not installed, then bit 24 of word 1 of the ORB must also be zero.

Invalid Sequence: The channel subsystem has fetched two successive CCWs both of which specify transfer in channel, or, depending on the model, a sequence of 256 or more CCWs with command chaining specified was executed by the channel subsystem and did not result in the transfer of any data to or from an I/O device.

Detection of the program-check condition during the initiation of an operation at the device causes the operation to be suppressed and the subchannel to be made status pending with primary, secondary, and alert status. When protection check is detected after the I/O operation has been initiated at the device, the device is signaled to conclude the operation after the available data logically prior to the protection check has been transferred. However, if an access violation occurs when the channel subsystem is in the process of fetching either a new IDAW or a new CCW while data chaining, and if the device signals the channel-end condition before transferring any data designated by the new CCW or IDAW, then the status is accepted, and the subchannel becomes status pending with primary and alert status and with protection check indicated. Other indications may accompany the protection-check indication as a function of the operation specified by the CCW, the status received from the device, and the current state of the subchannel. The protection-check condition causes command chaining and command retry to be suppressed.

Channel-Data Check
Channel-data check indicates that an uncorrected storage error has been detected in regard to data, contained in main storage, that is currently used in the performance of an I/O operation. The condition may be indicated when detected, even if the data is not used when prefetched. Channel-data check is indicated when data or the associated key has an invalid checking-block code (CBC) in main storage when that data is referenced by the channel subsystem.

On an input operation, when the channel subsystem attempts to store less than a complete checking block, and invalid CBC is detected on the checking block in storage, the contents of the location remain unchanged and with invalid CBC. On an output operation, whenever channel-data check is indicated, no bytes from the checking block with invalid CBC are transferred to the device.

During a storage access, the maximum number of bytes that can be transferred is model dependent. If a channel-data-check condition is recognized during that storage access, the number of bytes transferred to or from storage may not be detectable by the channel subsystem. Consequently, the number of bytes transferred to or from storage may not be correctly reflected by the residual count. However, the residual count that is stored in the SCSW, when used in conjunction with the
storage-access code and the CCW address, designates a byte location within the page in which the channel-data-check condition was recognized.

A condition indicated as channel-data check causes the current operation, if any, to be terminated. The subchannel becomes status pending with primary and alert status, or with primary, secondary, and alert status, as a function of the status received from the device. The count and address fields of the SCSW stored by TEST SUBCHANNEL pertain to the operation terminated. The extended-status-word-format bit is one, and subchannel-logout information is stored in the ESW, when TEST SUBCHANNEL is executed.

Whenever the channel-data-check condition pertains to prefetched data, the failing-storage-address-validity flag, bit 6 of the ERW, is one. An address of a location within the checking block for which the channel-data-check condition is generated is stored in the ESW failing-storage-address field.

Uncorrectable storage or key errors detected on prefetched data while the subchannel is start pending cause the operation to be canceled before initiation at the device. In this case, the subchannel is made status pending with primary, secondary, and alert status, with channel-data check indicated, and with the ESW failing-storage address stored.

Whenever channel-data check is indicated, no measurement data for the subchannel is stored.

**Channel-Control Check**

Channel-control check is caused by any machine malfunction affecting channel-subsystem controls. The condition includes invalid CBC on a CCW, an IDAW, or the respective associated key. The condition may be indicated when an invalid CBC is detected on a prefetched CCW, IDAW, or the respective associated key, even if that CCW or IDAW is not used.

Channel-control check may also indicate that an error has been detected in the information transferred to or from main storage during an I/O operation. However, when this condition is detected, the error has occurred inboard of the channel path: in the channel subsystem or in the path between the channel subsystem and main storage.

Detection of the channel-control-check condition causes the current operation, if any, to be terminated immediately. The subchannel is made status pending with primary and alert status or with primary, secondary, and alert status as a function of the type of termination, the current subchannel state, and the device status presented, if any. When the channel subsystem recognizes a channel-control-check condition, any previously existing intermediate interruption condition caused by a PCI flag in a CCW may or may not be recognized by the channel subsystem. The count and data-address fields of the SCSW stored by TEST SUBCHANNEL pertain to the operation terminated. The extended-status-word-format bit is one, and subchannel-logout information is stored in the ESW, when TEST SUBCHANNEL is executed.

Whenever the channel-control-check condition pertains to an invalid CBC detected on a prefetched CCW, a prefetched IDAW, or the key associated with the prefetched CCW or the prefetched IDAW, an extended-report word with bit 6 set to one, and the failing-storage address, are stored in the ESW when TEST SUBCHANNEL is executed.

Channel-control-check conditions encountered while prefetching when the subchannel is start pending cause the operation to be canceled before initiation at the device. In this case, the subchannel is made status pending with primary, secondary, and alert status, with channel-control check indicated, and with a failing-storage address that will be stored in the ESW.

If a subchannel is halt pending and the channel subsystem encounters a channel-control-check condition while performing the halt function for that subchannel, the subchannel remains halt pending unless the channel subsystem can determine that the halt signal was issued. The subchannel remains halt pending even if the channel subsystem was attempting to issue the halt signal and is unable to determine if the halt signal was issued.

If a subchannel is start pending or resume pending and the channel subsystem encounters a channel-control-check condition while performing the start function for that subchannel, the subchannel remains start pending or resume pending.
unless the channel subsystem can determine that the first command was accepted. The subchannel remains start pending or resume pending even if the channel subsystem was attempting to initiate the I/O operation for the first command and is unable to determine if the command was accepted. If the channel subsystem is unable to determine whether the first command was accepted, the subchannel is made status pending with at least alert and primary status.

In some situations in which a channel-subsystem malfunction exists, the channel-control-check condition may be reported as a machine-check condition.

Whenever channel-control check is indicated, no measurement data for the subchannel is stored.

**Programming Note:** If the status-control field of the SCSW indicates that the subchannel is status pending with alert status but the field-validity flags of the SCSW indicate that the device-status field is not usable for error-recovery purposes, the program should (1) assume that the channel-control-check condition occurred while the channel subsystem was accepting alert status from the device and (2) take the appropriate action for alert status, even though the status itself has been lost.

**Interface-Control Check**

Interface-control check indicates that an invalid signal has occurred on the channel path. The condition is detected by the channel subsystem and usually indicates malfunctioning of an I/O device. Interface-control check can occur for any of the following reasons:

1. A data or status byte received from a device while the subchannel is subchannel-and-device active or device active has an invalid checking-block code.
2. The status byte received from a device while the subchannel is idle, start pending, suspended, or halt pending has an invalid checking-block code.
3. A device responded with an address other than the address designated by the channel subsystem during initiation of an operation.
4. During command chaining, the device appeared not operational.
5. A signal from an I/O device either did not occur or occurred at an invalid time or had an invalid duration.
6. The channel subsystem recognized the I/O-error-alert condition (see "I/O-Error Alert (A)" on page 16-35).
7. ESW bit 26, indicating device-status check, is set to one.

Detection of the interface-control-check condition causes the current operation, if any, to be terminated immediately, and the subchannel is made status pending with alert status, primary and alert status, secondary and alert status, or primary, secondary, and alert status as a function of the type of termination, the current subchannel state, and the device status presented, if any. When the channel subsystem recognizes an interface-control-check condition, any previously existing intermediate interruption condition caused by a PCI flag in a CCW may or may not be recognized by the channel subsystem. The extended-status-word-format bit is one, and subchannel/logout information is stored in the ESW, when TEST SUBCHANNEL is executed.

If a subchannel is halt pending and the channel subsystem encounters an interface-control-check condition while performing the halt function for that subchannel, the subchannel remains halt pending unless the channel subsystem can determine that the halt signal was issued. The subchannel remains halt pending even if the channel subsystem was attempting to issue the halt signal and is unable to determine if the halt signal was issued.

If a subchannel is start pending or resume pending and the channel subsystem encounters an interface-control-check condition while performing the start function for that subchannel, the subchannel remains start pending or resume pending unless the channel subsystem can determine that the first command was accepted. The subchannel remains start pending or resume pending even if the channel subsystem was attempting to initiate the I/O operation for the first command and is unable to determine if the command was accepted. If the channel subsystem is unable to determine whether the first command was accepted, the subchannel is made status pending with at least alert and primary status.
If, while initiating a signaling sequence with the channel subsystem for the purpose of presenting status or transferring data, the device presents an address with invalid parity, the error condition is not made available to the program since the identity of the device and associated subchannel are unknown.

Whenever interface-control check is indicated, no measurement data for the subchannel is stored.

**Programming Note:** If the status-control field of the SCSW indicates that the subchannel is status pending with alert status but the field-validity flags of the SCSW indicate that the device-status field is not usable for error-recovery purposes, the program should (1) assume that the interface-control-check condition occurred while the channel subsystem was accepting alert status from the device and (2) take the appropriate action for alert status, even though the status itself has been lost.

**Chaining Check**
Chaining check is caused by channel-subsystem overrun during data chaining on input operations. The condition occurs when the I/O-data rate is too high for the particular resolution of data addresses. Chaining check cannot occur on output operations.

Detection of the chaining-check condition causes the I/O device to be signaled to conclude the operation. It causes command chaining to be suppressed.

**Count Field**
Bit positions 16-31 of word 2 contain the residual count. The count is to be used in conjunction with the original count specified in the last CCW and, depending upon existing conditions (see Figure 16-4 on page 16-19), indicates the number of bytes transferred to or from the area designated by the CCW. The count field is meaningful whenever the subchannel is status pending with primary status that consists of either (1) device status only or (2) device status together with subchannel status of incorrect length only, PCI only, or both.

In Figure 16-5 on page 16-30 the contents of the count field are listed for all cases where the subchannel is start pending, subchannel-and-device active, device active, suspended, or status pending.
<table>
<thead>
<tr>
<th>Subchannel State</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start pending (UUU0/AIPSX)</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Start pending and status pending (10YY1/AIPSX)</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Start pending and status pending (00111/AIPSX) because the device appeared not operational on all paths</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Start pending and device active (UUUU0/AIPSX)</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Suspended (YYYY/Y/AIPSX)</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Subchannel-and-device active (UUUU0/AIPSX)</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Device active (UUUU0/AIPSX)</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Status pending (01001/AIPSX) because of program-controlled-interruption condition or initial-status interruption</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Status pending (1111/AIPSX); termination occurred because of:</td>
<td></td>
</tr>
<tr>
<td>Program check</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Protection check</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Chaining check</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Channel-control check</td>
<td>See note 1</td>
</tr>
<tr>
<td>Interface-control check</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Channel-data check</td>
<td>See note 2</td>
</tr>
<tr>
<td>Status pending (Y111/AIPSX); termination occurred under count control</td>
<td>Correct</td>
</tr>
<tr>
<td>Status pending (Y001/AIPSX)</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Status pending (1111/AIPSX)</td>
<td>Correct; residual count of last used CCW</td>
</tr>
<tr>
<td>Status pending (1111/AIPSX); command chaining suppressed because of alert status</td>
<td>Correct; residual count of last used CCW</td>
</tr>
<tr>
<td>Status pending (YYYY1/AIPSX); after termination by HALT SUBCHANNEL</td>
<td>Unpredictable</td>
</tr>
<tr>
<td>Status pending (00001/AIPSX); after termination by CLEAR SUBCHANNEL</td>
<td>Not meaningful</td>
</tr>
<tr>
<td>Status pending (YY111/AIPSX); operation completed normally at the subchannel</td>
<td>Correct; indicates the residual count</td>
</tr>
</tbody>
</table>

*Figure 16-5 (Part 1 of 2). Contents of Count Field in the SCSW*
<table>
<thead>
<tr>
<th>Subchannel State1</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status pending (1Y111/AIPSX); command chaining terminated because of alert status2</td>
<td>Correct; original count of CCW specifying the new I/O operation</td>
</tr>
<tr>
<td>Status pending (10001/AIPSX) because of alert status</td>
<td>Not meaningful3</td>
</tr>
</tbody>
</table>

**Explanation:**

1. In situations where more than a single condition exists because of, for example, alert status that is described by program check and unit check, the entry appearing first in the table takes precedence.

The meaning of the notation in this column is as follows:

- A Alert status
- I Intermediate status
- P Primary status
- S Secondary status
- X Status pending

The allowed combination of status-control-bit settings is shown to the left of the "/" symbol.

Bit settings are specified as follows:

- 0 Corresponding condition is not indicated.
- 1 Corresponding condition is indicated.
- U Unpredictable. The corresponding condition is not meaningful when the subchannel is not status pending.
- Y Corresponding condition is not significant and is indicated as a function of the subchannel state.

2. The subchannel may also be resume pending.

3. The contents of the count field are not meaningful because the count field is not valid when the SCSW is stored and the subchannel is in the given state.

**Notes:**

1. The count is unpredictable unless IDAW check is indicated, in which case the count may not correctly reflect the number of bytes transferred to or from main storage but will (when used in conjunction with the CCW address) designate a byte location within the page in which the channel-control-check condition was recognized.

2. During a storage access, the maximum number of bytes that can be stored by a channel subsystem is model dependent. If a channel-data-check condition is recognized during that access, the number of bytes transferred to or from storage may not be detectable by the channel subsystem. Consequently, the number of bytes transferred to or from storage may not be correctly reflected by the residual count. However, the residual count that is stored when used in conjunction with the storage-access code and the CCW address designates a byte location within the page in which the channel-data-check condition was recognized.

Figure 16-5 (Part 2 of 2). Contents of Count Field in the SCSW
Extended-Status Word

The extended-status word (ESW) provides additional information to the program about the subchannel and its associated device. The ESW is placed in words 3-7 of the IRB designated by the second operand of TEST SUBCHANNEL when TEST SUBCHANNEL is executed and the subchannel designated is operational. If the subchannel is status pending or status pending with any combination of primary, secondary, intermediate, or alert status (except as noted in the next paragraph) when TEST SUBCHANNEL is executed, the ESW may have any of the following types of extended-status format:

Format 0 Subchannel logout in word 0, an ERW in word 1, a failing-storage address or zeros in words 2 and 3, and a secondary-CCW address or zeros in word 4.

Format 1 Zeros in bytes 0, 2, and 3 of word 0, the LPUM in byte 1 of word 0, an ERW in word 1, and zeros in words 2-4.

Format 2 Zeros in byte 0, the LPUM in byte 1, and the device-connect time in bytes 2 and 3 of word 0; an ERW in word 1; zeros in words 2-4.

Format 3 Zeros in byte 0, the LPUM in byte 1, and unpredictable values in bytes 2 and 3 of word 0; an ERW in word 1; zeros in words 2-4.

Words 0-4 of the ESW contain unpredictable values if any of the following conditions is met:

1. The subchannel is not status pending.
2. The subchannel is status pending alone, and the extended-status-word-format bit is zero.
3. The subchannel is status pending with intermediate status alone for other than the intermediate interruption condition due to suspension.

The type of extended-status format stored depends upon conditions existing at the subchannel at the time TEST SUBCHANNEL is executed. The conditions under which each of the types of formats is stored are described in the remainder of this section.

Extended-Status Format 0

The ESW stored by TEST SUBCHANNEL is a format-0 ESW when the extended-status-word-format bit, bit 5 of word 0 of the SCSW, is one and the subchannel is status pending with any combination of status as defined in Figure 16-6 on page 16-36. In this case, subchannel-logout information and an ERW are stored in the extended-status word. Subchannel logout provides detailed model-independent information relating to a subchannel and describing equipment errors detected by the channel subsystem. The information is provided to aid the recovery of an I/O operation, a device, or both. Whenever subchannel logout is provided, the error conditions relate only to the subchannel reporting the error. If I/O operations involving other subchannels have been affected by the error condition, those subchannels also provide similar subchannel-logout information. An extended-report word provides additional information relating to the cause of the malfunction.

A format-0 ESW has the following format:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subchannel Logout</td>
<td>Extended-Report Word</td>
<td>Failing-Storage Address or ESLD</td>
<td>Secondary-CCW Address</td>
<td></td>
</tr>
</tbody>
</table>
```

Extended-Status Flags (ESF): Any of bits 1-7, when one, specifies that an error-check condition has been detected by the channel subsystem. The following indications are provided in the ESF field:

**Key Check.** Bit 1, when one, indicates that the channel subsystem, when accessing data, when attempting to update the measurement block, or when attempting to fetch either a
CCW or an IDAW, has detected an invalid checking-block code (CBC) on the associated storage key. The channel-data-check bit, bit 12 of word 2 of the SCSW, the measurement-block data-check bit, bit 3 of word 0 of the ESW, the CCW-check bit, bit 5 of word 0 of the ESW, or the IDAW-check bit, bit 6 of word 0 of the ESW, identifies the source of the key error.

**Note:** This condition may be indicated to the program when an invalid checking-block code on a key is detected but the data, CCW, or IDAW then is not used after being prefetched. In this case, the failing-storage-address-validity bit, bit 6 of the ERW, is one, indicating that the address of a location within the storage block having the key is stored in words 2 and 3 of the ESW.

*Measurement-Block Program Check.* Bit 2, when one, indicates that the channel subsystem, in attempting to update the measurement block, has either detected an invalid absolute address when combining the measurement-block origin with the measurement-block index for this subchannel or has detected an invalid measurement-block address at the subchannel.

*Measurement-Block Data Check.* Bit 3, when one, indicates that a malfunction has been detected involving the data of the measurement block in main storage. (See “Measurement Block” on page 17-3) Measurement-block data check is indicated when the measurement block is updated and an invalid checking-block code (CBC) is detected on the storage used to contain the measurement data or on the associated key. When invalid CBC on the associated key is detected, the key-check bit, bit 1 of the ESF field, is also stored as one.

*Measurement-Block Protection Check.* Bit 4, when one, indicates that the channel subsystem, when attempting to update the measurement block, has been prohibited from accessing the measurement block because the storage key does not match the measurement-block key (see “Measurement Block” on page 17-3). The key provided by SET CHANNEL MONITOR is used for the access of storage associated with measurement-block-update operations (see “SET CHANNEL MONITOR” on page 14-12).

**Note:** Whenever any of the measurement-check conditions is indicated by bits 2-4, the channel subsystem sets the subchannel measurement-block-update-enable bit to zero, disabling the storing of measurement data for the subchannel (see “Measurement-Mode Enable (MM)” on page 15-3).

*CCW Check.* Bit 5, when one, indicates that an invalid CBC on the contents of the CCW or its associated key has been detected. When either of these conditions is detected, the I/O operation is terminated, the subchannel becomes status pending with primary and alert status, the extended-status-word-format bit in the SCSW is stored as one, and channel-control check is indicated in the subchannel-status field. The subchannel also becomes status pending with secondary status as a function of the type of termination or status received from the device. When invalid CBC on the associated key is detected, the key-check bit, bit 1 of the ESF field, is also stored as one.

**Note:** This condition may be indicated to the program when an invalid checking-block code on a prefetched CCW is detected but the CCW is not used. In this case, the failing-storage-address-validity bit, bit 6 of the ERW, is one, indicating that the address of a location having the invalid CBC is stored in words 2 and 3 of the ESW.

*IDAW Check.* Bit 6, when one, indicates that an invalid CBC on the contents of an IDAW or its associated key has been detected. When either of these conditions is detected, the I/O operation is terminated with the device, the subchannel becomes status pending with primary and alert status, the extended-status-word-format bit in the SCSW is one, and channel-control check is indicated in the subchannel-status field. The subchannel also becomes status pending with secondary status as a function of the type of termination or status received from the device. When invalid CBC on the associated key is detected, the key-check bit, bit 1 of the ESF field, is also one.

**Note:** This condition may be indicated to the program when an invalid checking-block code on the contents of a prefetched IDAW is detected but the IDAW is not used. In this
case, the failing-storage-address-validity bit, bit 6 of the ERW, is one, indicating that the address of a location having the invalid CBC is stored in words 2 and 3 of the ESW. Detection of a channel-data-check condition does not cause the CCW-check and IDAW-check bits to be stored as ones.

Reserved. Bit 7 is stored as zero.

**Last-Path-Used Mask (LPUM):** Bits 8-15 indicate the channel path that was last used for communicating or transferring information between the channel subsystem and the device. The bit corresponding to the channel path in use is set whenever any of the following occurs:

1. The first command of a start-subchannel function is accepted by the device (see "Activity Control (AC)" on page 16-13).
2. The device and channel subsystem are actively communicating when the channel subsystem performs the suspend function for the channel program in execution.
3. The channel subsystem accepts status from the device that is recognized as an interruption condition, or a condition has been recognized that suppresses command chaining (see "Suppression Conditions" on page 16-2).
4. The channel subsystem recognizes an interface-control-check condition (see "Interface-Control Check" on page 16-28), and no subchannel-logoff information is currently present at the subchannel.

The LPUM field contains the most recent setting and is valid whenever the ESW contains information in one of the formats 0-3 (see "Extended Status Word" on page 16-32) and the SCSW is stored. When subchannel-logoff information is present in the ESW, a zero LPUM-field-validity flag indicates that the LPUM setting is not consistent with the other subchannel-logoff indications.

**Ancillary Report (R):** Bit 16, when one, indicates that a malfunction of a system component has occurred that has been recognized previously or that has affected the activities of multiple subchannels. When the malfunction affects the activities of multiple subchannels, an ancillary-report condition is recognized for all of the affected subchannels except one. This bit, when zero, indicates that this malfunction of a system component has not been recognized previously. This bit is meaningful only when a channel-control check, channel-data check, or an interface-control check is indicated in bit positions 12-14 of word 2 of the SCSW.

Depending on the model, recognition of an ancillary-report condition may not be provided or it may not be provided for all system malfunctions that effect subchannel activity. When ancillary-report recognition is not provided, bit 16 is set to zero.

**Field-Validity Flags (FVF):** Bits 17-21 indicate the validity of the information stored in the corresponding fields of either the SCSW or the extended-status word. When the validity bit is one, the corresponding field has been stored and is usable for recovery purposes. When the validity bit is zero, the corresponding field is not usable.

This bit-significant field has meaning when channel-data check, channel-control check, or interface-control check is indicated in the SCSW. When these checks are not indicated, this field, as well as the termination-code and sequence-code fields, has no meaning. Furthermore, when these checks are not indicated, the last-path-used-mask, device-status, and CCW-address fields are all valid. The fields are defined as follows:

17. Last-path-used mask
18. Termination code
19. Sequence code
20. Device status
21. CCW address

**Storage-Access Code (SA):** Bits 22-23 indicate the type of storage access that was being performed by the channel subsystem at the time of error. The SA field pertains only to the access of storage for the purpose of fetching or storing data during the performance of an I/O operation. This encoded field has meaning only when channel-data check, channel-control check, or interface-control check is indicated in the subchannel status. The access-code assignments are as follows:

00. Access type unknown
01. Read
10. Write
11. Read backward
Termination Code (TC): Bits 24 and 25 indicate the type of termination that has occurred. This encoded field has meaning only when channel-data check, channel-control check, or interface-control check is indicated in the SCSW. The types of termination are as follows:

00  Halt signal issued
01  Stop, stack, or normal termination
10  Clear signal issued
11  Reserved

When at least one channel check is indicated in the SCSW but the termination-code-field-validity flag is zero, it is unpredictable which, if any, termination has been signaled to the device. If more than one channel-check condition is indicated in the SCSW, the device may have been signaled one or more termination codes that are the same or different. In this situation, if the termination-code-field-validity flag is one, the termination code indicates the most severe of the terminations signaled to the device. The termination codes, in order of increasing severity, are: stop, stack, or normal termination (01); halt signal issued (00); and clear signal issued (10).

Device-Status Check (D): When the status-verification facility is installed, bit 26, when one, indicates that the subchannel logout in the ESW resulted from the channel subsystem detecting device status that had valid CBC but that contained a combination of bits that was inappropriate when the status byte was presented to the channel subsystem. When the device-status-check bit is one, the interface-control-check status bit is set to one. If, additionally, bit 20 of the subchannel-logout field has been stored as one, then the status byte in error has been stored in the device-status field of the SCSW. If the status-verification facility is not installed, bit 26 is stored as zero.

Secondary Error (E): Bit 27, when one, indicates that a malfunction of a system component that may or may not have been directly related to any activity involving subchannels or I/O devices has occurred. Subsequent to this occurrence, the activity related to this subchannel and the associated I/O device was affected and caused the subchannel to be set status pending with either channel-control check or interface-control check.

I/O-Error Alert (A): Bit 28, when one, indicates that subchannel logout in the ESW resulted from the signaling of I/O-error alert. The I/O-error-alert signal indicates that the control unit or device has detected a malfunction that must be reported to the channel subsystem. The channel subsystem, in response, issues a clear signal and, except as described in the next paragraph, causes interface-control check to be set and extended-status-format-0 (logout) information to be stored in the ESW.

When I/O-error alert is signaled and the subchannel has previously been set disabled or no subchannel is associated with the device, the clear signal is issued to the device, and the I/O-error-alert indication is ignored by the channel subsystem.

Sequence Code (SC): Bits 29-31 identify the I/O sequence in progress at the time of error. The sequence code pertains only to I/O operations initiated by the execution of START SUBCHANNEL or RESUME SUBCHANNEL. This encoded field has meaning only when channel-data check, channel-control check, or interface-control check is indicated in the SCSW.

The sequence-code assignments are:

000  Reserved.
001  A nonzero command byte has been sent by the channel subsystem, but a response has not yet been analyzed by the channel subsystem. This code is set during the initiation sequence.
010  The command has been accepted by the device, but no data has been transferred.
011  At least one byte of data has been transferred between the channel subsystem and the device. This code may be used when the channel path is in an idle or poll state.
100  The command in the current CCW (1) has not yet been sent to the device, (2) was sent but not accepted by the device, or (3) was sent and accepted but command-retry status was presented. This code is set when any of the following conditions occurs:

1. The command address is updated during command chaining or during the initiation of a start function or resume function at the device.
2. During the initiation sequence, the status includes attention, control unit end, unit check, unit exception, busy, status modifier (without channel end and device end), or device end (without channel end).

3. Command retry is signaled.

4. The channel subsystem interrogates the device in the process of clearing an interruption condition.

5. The channel subsystem signals the conclusion of the chain of operations to the device during command chaining while performing the suspend function.

101 The command in the current CCW has been accepted, but data transfer is unpredictable. This code applies from the time a device is logically connected to a channel path until the time it is determined that a new sequence code applies. This code may also be used when the channel subsystem places a channel path in the polling or idle state and it is impossible to determine that code 010 or 011 applies. It may also be used at other times when a channel path cannot distinguish between code 010 or 011.

110 Reserved.

111 Reserved.

Figure 16-6 defines the relationship between indications provided as subchannel-logout data and the appropriate SCSW bits.

101 The command in the current CCW has been accepted, but data transfer is unpredictable. This code applies from the time a device is logically connected to a channel path until the time it is determined that a new sequence code applies. This code may also be used when the channel subsystem places a channel path in the polling or idle state and it is impossible to determine that code 010 or 011 applies. It may also be used at other times when a channel path cannot distinguish between code 010 or 011.

110 Reserved.

111 Reserved.

Figure 16-6 defines the relationship between indications provided as subchannel-logout data and the appropriate SCSW bits.

Extended-Report Word

The extended-report word (ERW) provides information to the program describing specific conditions that may exist at the device, subchannel, or channel subsystem. The ERW is stored whenever the extended-status word is stored. When the extended-status-word-format bit, bit 5 of word 0 of the SCSW, and the extended-control bit, bit 14 of word 0 of the SCSW, are both zeros, the ERW contains all zeros. When the extended-status-word-format bit or the extended-control bit or both are ones, the ERW has the following format:

<table>
<thead>
<tr>
<th>0</th>
<th>L</th>
<th>E</th>
<th>A</th>
<th>P</th>
<th>T</th>
<th>F</th>
<th>S</th>
<th>C</th>
<th>R</th>
<th>SCNT</th>
<th>Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td>16</td>
<td>16</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Request Logging Only (L): Bit 1, when one, requests that the program only record information about the condition that caused the extended-status word to be stored.
Extended-Subchannel-Logout Pending (E): Bit 2, when one, and when the extended-status-word-format bit, bit 5 of word 0 of the SCSW, is also one, indicates that an extended-subchannel logout is pending and that words 2-3 of the ESW contain the extended-subchannel-logout descriptor.

Bit 2 is set to zero when bit 6 is set to one.

When an ERW is stored with bits 2 and 6 both set to zero, zeros are stored in words 2 and 3 of the ESW.

Authorization Check (A): Bit 3, when one, indicates that the start or resume function was terminated because the channel subsystem has been placed in the isolated state in which pending I/O operations are not initiated and I/O operations currently being performed either are in the process of being terminated or have been terminated.

Path Verification Required (P): Bit 4, when one, indicates that the program must verify the identity of the device. The LPUM, when valid, indicates the channel path for which device verification is to be performed. When a valid LPUM is not available, the identity of the device must be verified for each available channel path.

Channel-Path Timeout (T): Bit 5, when one, indicates that, during a signaling sequence, an appropriate signal from the device did not occur within a predetermined time interval. Bit 5 is meaningful when the extended-status-word-format bit, bit 5 of word 0 of the SCSW, and the interface-control-check bit, bit 14 of word 2 of the SCSW, are both ones.

Failing-Storage-Address Validity (F): Bit 6, when one, and when the extended-status-word-format bit, bit 5 of word 0 of the SCSW, is also one, indicates that the channel subsystem has detected an invalid CBC on a CCW, a data location, an IDAW, or the respective associated key and has stored, in words 2 and 3 of the ESW, the absolute address of a location associated with the invalid CBC.

Bit 6 is set to zero when bit 2 is set to one.

When an ERW is stored with bits 2 and 6 both set to zero, zeros are stored in words 2 and 3 of the ESW.

Concurrent Sense (S): Bit 7, when one, indicates that the concurrent-sense facility has placed sense information accepted from the device in the extended-control word and has stored a value, in bit positions 10-15 of the ERW, that specifies the number of sense bytes that have been stored in the extended-control word. When bit 7 is one, bit 14 of word 0 of the SCSW is also one.

Concurrent-Sense Count (SCNT): When bit 7 is one, bit positions 10-15 contain a value, in the range 1-32, that specifies the number of sense bytes stored into the extended-control word by the concurrent-sense facility. When bit 7 is zero, bit positions 10-15 contain zeros.

Secondary-CCW-Address Validity (C): Bit 8, when one, and when the extended-status-word-format bit, bit 5 of word 0 of the SCSW, is also one, indicates that the channel subsystem has detected an error condition that precludes the continued performance of an I/O operation. When prefetching applies (bit 9 of word 1 of the ORB is one) and certain error conditions identified by channel-control check, channel-data check, or interface-control check are recognized by the channel subsystem, situations may exist where the termination point of execution of the channel program differs between the channel subsystem and the I/O device. To properly identify the termination points, bit 8 is set to one, and a second CCW address (secondary-CCW address) is provided in the ESW and designates the last CCW executed at the device. When the validity bit is zero for the previously mentioned errors, the channel subsystem was unable to determine the termination point of the control-unit execution, and the secondary-CCW-address field contains zeros.

Bit 8 is not set to one unless the program has permitted prefetching by setting the prefetch-control bit, bit 9 of word 1 of the ORB, to one for the channel program in execution.

Failing-Storage-Address Format (R): Bit 9 indicates the format of the failing-storage address when the failing-storage-address validity bit, bit 6 of the ERW, is one. When bit 6 is zero, bit 9 is not meaningful and is stored as zero. When bit 6 is one and bit 9 is zero, a format-1 failing-storage address is stored in words 2 and 3 of the ESW. When bit 6 is one and bit 9 is one, a format-2 failing-storage address is stored in words 2 and 3.
See “Failing-Storage Address” below for a description of format-1 and format-2 addresses.

**Failing-Storage Address**
Words 2 and 3 of the extended-status contain a 24-, 31-, or 64-bit absolute address. When the failing-storage-address-validity flag, bit 6 of the ERW, is one, words 2 and 3 contain either a format-1 failing-storage address or a format-2 failing-storage address. When bit 6 is one, the failing-storage-address field designates a byte location within the invalid checking block associated with an invalid CBC for a CCW, data location, IDAW, or their respective associated key.

The form of the address stored in words 2 and 3 depends on the format-2-IDAW control, bit 14 of word 1 of the ORB, when an error condition is detected. When the control is zero, specifying that fullword IDAWs containing 31-bit addresses are used, a format-1 address is stored, and the failing-storage-address-format bit, bit 9 of the ERW, is stored as zero. When the format-2-IDAW control is one, specifying that doubleword IDAWs containing 64-bit addresses are used, a format-2 address is stored, and the failing-storage-address-format bit is stored as one.

When a format-1 address is stored, bits 1-31 of word 2 form the address associated with the reported error condition, and bit 0 of word 2 and all of word 3 are stored as zeros. When a format-2 address is stored, bits 0-31 of word 2 followed by bits 0-31 of word 3 form the 64-bit address associated with the reported error condition.

**Extended-Subchannel-Logout Descriptor (ESLD)**

When the extended-status-word-format (L) bit, bit 5 of SCSW word 0, is one and the extended-subchannel-logout pending (E) flag, bit 2 of the ERW, is one, an extended-subchannel-logout descriptor is provided in words 2-3 of the ESW. The ESLD identifies the logout and the channel path for which the extended-subchannel logout is pending.

**Secondary-CCW Address**
When the subchannel-status field indicates channel-control check, channel-data check, or interface-control check and the secondary-CCW-address-validity flag, bit 8 of word 1, is one, bits 1-31 of word 4 form an absolute address of the last CCW executed by the I/O device at the point the reported check condition caused channel-program termination. When provided, the secondary-CCW address may be used for recovery purposes. When the secondary-CCW-address-validity flag is zero, this field contains zeros.

**Extended-Status Format 1**
The ESW stored by TEST SUBCHANNEL is a format-1 ESW when all of the following conditions are met:

1. The extended-status-word-format bit, bit 5 of word 0 of the SCSW, is zero.
2. The subchannel status-control field has the status-pending bit, bit 31 of word 0 of the SCSW, set to one, together with:
   a. The primary-status bit, bit 29 of word 0 of the SCSW, alone,
   b. The primary-status bit and other status-control bits, or
   c. The intermediate-status bit, bit 28 of word 0 of the SCSW, and the suspended bit, bit 26 of word 0 of the SCSW.
3. At least one of the following conditions is indicated:
   a. The device-connect-time-measurement mode is inactive.
   b. The channel-subsystem-timing facility is not available for the subchannel.
   c. The subchannel is not enabled for the device-connect-time-measurement mode.

Zeros are stored in bytes 0, 2, and 3 of word 0, and the LPUM is stored in byte 1 of word 0; an ERW is stored in word 1; zeros are stored in words 2-4.

The device-connect-time-measurement mode is made inactive when SET CHANNEL MONITOR is executed and bit 31 of general register 1 is zero.

A format-1 ESW has the following format:
**Last-Path-Used Mask (LPUM):** For a definition of the LPUM, see “Last-Path-Used Mask (LPUM)” on page 16-34.

**Extended-Report Word (ERW):** For a definition of the ERW, see “Extended-Report Word” on page 16-36.

**Extended-Status Format 2**

The ESW stored by TEST SUBCHANNEL is a format-2 ESW when all of the following conditions are met:

1. The extended-status-word-format bit, bit 5 of word 0 of the SCSW, is zero.
2. The channel-subsystem-timing facility is available for the subchannel.
3. The subchannel is enabled for the device-connect-time-measurement mode.
4. The device-connect-time-measurement mode is active.
5. The subchannel status-control field has the status-pending bit, bit 31 of word 0 of the SCSW, set to one, together with:
   a. The primary-status bit, bit 29 of word 0 of the SCSW, alone,
   b. The primary-status bit and other status-control bits, or
   c. The intermediate-status bit, bit 28 of word 0 of the SCSW, and the suspended bit, bit 26 of word 0 of the SCSW.

Zeros are stored in byte 0 of word 0, the LPUM is stored in byte 1 of word 0, and the device-connect time is stored in bytes 2 and 3 of word 0; an ERW is stored in word 1; zeros are stored in words 2-4.

A format-2 ESW has the following format:

<table>
<thead>
<tr>
<th>0</th>
<th>Zeros</th>
<th>LPUM</th>
<th>Zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Extended-Report Word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Zeros</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Device-Connect-Time Interval (DCTI):** Bit positions 16-31 contain the binary count of time increments accumulated by the channel subsystem during the time that the channel subsystem and the device were actively communicating and the subchannel was subchannel active. The time increment of the DCTI is 128 microseconds.

If the above conditions for the storing of the DCTI value in the ESW are met but the device-connect-time-measurement mode was made active by SET CHANNEL MONITOR subsequent to the execution of START SUBCHANNEL for this subchannel, the DCTI value stored is greater than or equal to zero and less than or equal to the correct DCTI value.

**Note:** The DCTI value stored in the ESW is the same as that used to update the corresponding measurement-block data for the subchannel if the measurement-block-update mode is in use for the subchannel. If the measurement-block-update mode for the channel subsystem is active and the subchannel is enabled for the device-connect-time-measurement mode but no DCTI value is stored in the ESW (because of the presence of subchannel-logout information), or if the DCTI is zero, then nothing is added to the corresponding measurement-block data.

**Extended-Report Word (ERW):** For a definition of the ERW, see “Extended-Report Word” on page 16-36.
Extended-Status Format 3

The ESW stored by TEST SUBCHANNEL is a format-3 ESW when the extended-status-word-format bit, bit 5 of word 0 of the SCSW, is zero and the subchannel is status pending with (1) secondary status, alert status, or both when primary status is not also present, or (2) intermediate status when the subchannel is not suspended. Zeros are stored in byte 0 of word 0, and the LPUM is stored in byte 1 of word 0. Bytes 2 and 3 of word 0 contain unpredictable values; an ERW is stored in word 1; zeros are stored in words 2-4.

A format-3 ESW has the following format:

```
0  Zeros   LPUM   XXXXXXX | XXXXXXX
1            Extended-Report Word
2            Zeros
3            
4            
```

Last-Path-Used Mask (LPUM): For a definition of the LPUM, see "Last-Path-Used Mask (LPUM)" on page 16-34.

An "X" in the format indicates the bit may be zero or one.


Figure 16-7 summarizes the conditions at the subchannel under which each type of information is stored in the ESW.

Extended-Control Word

The extended-control word, which is words 8-15 of an interruption-response block (see "Interruption-Response Block" on page 16-6), provides additional information to the program describing conditions that may exist at the channel subsystem, subchannel, or device. The extended-control (E) bit, bit 14 of word 0 of the SCSW, when one, indicates that model-dependent information or concurrent-sense information has been stored in the extended-control word.

The information provided in the extended-control word is as follows:

---0  
00001 0  
01001 0  
**=1=1 0  
**=011 0  
1=001 0  
***=1 1  

Explanation:
- Defined to be not meaningful when X is zero.
- Bits may be zeros or ones.
- Information not relevant in this situation.
- Alert status.
- D Accumulated device-connect-time-interval (DCTI) value stored in bytes 2 and 3.
- I Intermediate status.
- L Extended-status-word format.
- M Last-path-used mask (LPUM) stored in byte 1.
- P Primary status.
- R Subchannel-logout information stored in word 0.
- S Secondary status.
- U No format defined.
- X Status pending.
- Z Bits are stored as zeros.
Extended-Measurement Word

The extended-measurement word (EMW) provides I/O measurement information to the program for the most recent start or resume operation performed at the subchannel. When the extended-I/O-measurement-word facility is installed and enabled, the EMW is conditionally stored in words 16-23 of the IRB designated by the second operand of TEST SUBCHANNEL when TEST SUBCHANNEL is executed. The EMW is stored by TEST SUBCHANNEL when all of the following conditions are met:

1. The extended-status-word-format (L) bit (bit 5, word 0 of the SCSW) is zero.
2. The channel-subsystem-timing facility is available for the subchannel, as indicated by the timing facility bit (T) at the subchannel.
3. The extended-I/O-measurement-word-mode enable bit (X) at the subchannel is one.
4. The subchannel status-control field has the status-pending bit (bit 31, word 0 of the SCSW) set to one, together with:
   a. The primary-status bit (bit 29, word 0 of the SCSW) alone, or
   b. The primary-status bit and other status-control bits, or
   c. The secondary-status bit (bit 29, word 0 of the SCSW) alone, or
   d. The intermediate-status bit (bit 28, word 0 of the SCSW) and the suspended bit (bit 26, word 0 of the SCSW).

Words 16-23 of the IRB contain unpredictable values when the extended-I/O-measurement-word facility is installed and enabled but the conditions listed above are not met.

When the extended-I/O-measurement-word facility is not installed, or the facility is installed but not enabled, words 16-23 of the IRB are not accessed by the channel subsystem.

The format of the extended-measurement word is shown below:

<table>
<thead>
<tr>
<th>Word</th>
<th>Device-Connect Time</th>
<th>Function-Pending Time</th>
<th>Device-Disconnect Time</th>
<th>Control-Unit-Queuing Time</th>
<th>Device-Active-Only Time</th>
<th>Device-Busy Time</th>
<th>Initial Command Response Time</th>
<th>reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each field in the EMW, when valid, contains a 32 bit binary count in which each increment of the count represents a value of 0.5 microseconds. A value of 00000000 hex represents a time period of zero seconds; the maximum value of FFFFFFFF hex represents approximately 35.79 minutes.

The accuracy of each of the measurement fields stored by the measurement facility is undefined and may vary depending on the resolution of timers implemented at the channel subsystem, the types of channels used to perform the operation, the capabilities of control units accessed during the operation, and the overall length of the I/O operation that was performed. The maximum value of FFFFFFFF hex is stored if a counter overflows; the program is not alerted when an overflow occurs.
Device-Connect Time: Bit positions 0-31 of word 0 contain the measured device-connect time for the operation. The device-connect time is the sum of the time intervals measured whenever the device is logically connected to a channel path while the subchannel is subchannel active and the device is actively communicating with the channel path, as defined in the section “Device-Connect Time” on page 17-5.

Function-Pending Time: Bit positions 0-31 of word 1 contain the SSCH- or RSCH-function-pending time for the operation. Function-pending time is the time interval between acceptance of the start function (or resume function if the subchannel is in the suspended state) at the subchannel and acceptance of the first command associated with the initiation or resumption of channel-program execution at the device, as defined in the section “Function-Pending Time” on page 17-5.

Device-Disconnect Time: Bit positions 0-31 of word 2 contain the device-disconnect time for the operation. Device-disconnect time is the sum of the time intervals measured whenever the device is logically disconnected from the channel subsystem during an I/O operation while the device is busy with an operation initiated from a different system, as defined in the section “Device-Disconnect Time” on page 17-5.

Control-Unit-Queuing Time: Bit positions 0-31 of word 3 contain the control-unit-queuing time for the operation. Control-unit-queuing time is the sum of the time intervals measured by the control unit whenever the device is logically disconnected from the channel subsystem during an I/O operation while the device is busy with an operation initiated from a different system, as defined in the section “Control-Unit-Queuing Time” on page 17-6.

Device-active-only time: Bit positions 0-31 of word 4 contain the device-active-only time for the operation. Device-active-only time is the sum of the time intervals when the subchannel is device-active but not subchannel-active at the end of an I/O operation or chain of I/O operations initiated by a start function or resume function, as defined in section “Device-Active-Only Time” on page 17-6.

Device-busy time: Bit positions 0-31 of word 5 contain the device-busy time for the operation. Device-busy time is the sum of the time intervals when the device is found to be device busy during an attempt to initiate a start function or resume function at the subchannel, as defined in the section “Device-Busy Time” on page 17-6.

Initial Command Response Time: Bit positions 0-31 of word 6 contain the initial-command-response time for the operation. The initial-command-response time for an operation is the time interval beginning from when the first command of the channel program is sent to the device until the device indicates it has accepted the command.
Chapter 17. I/O Support Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter-Subsystem Monitoring</td>
<td>17-1</td>
</tr>
<tr>
<td>Channel-Subsystem Timing</td>
<td>17-2</td>
</tr>
<tr>
<td>Channel-Subsystem Timer</td>
<td>17-2</td>
</tr>
<tr>
<td>Measurement-Block Update</td>
<td>17-3</td>
</tr>
<tr>
<td>Measurement Block</td>
<td>17-3</td>
</tr>
<tr>
<td>Measurement-Block Format</td>
<td>17-7</td>
</tr>
<tr>
<td>Measurement-Block Origin</td>
<td>17-7</td>
</tr>
<tr>
<td>Measurement-Block Address</td>
<td>17-8</td>
</tr>
<tr>
<td>Measurement-Block Key</td>
<td>17-8</td>
</tr>
<tr>
<td>Measurement-Block Index</td>
<td>17-8</td>
</tr>
<tr>
<td>Measurement-Block-Update Mode</td>
<td>17-8</td>
</tr>
<tr>
<td>Measurement-Block-Update Enable</td>
<td>17-9</td>
</tr>
<tr>
<td>Control-Unit-Queuing Measurement</td>
<td>17-9</td>
</tr>
<tr>
<td>Control-Unit-Defer Time</td>
<td>17-9</td>
</tr>
<tr>
<td>Device-Active-Only Measurement</td>
<td>17-9</td>
</tr>
<tr>
<td>Initial-Command-Response Measurement</td>
<td>17-10</td>
</tr>
<tr>
<td>Time-Interval-Measurement Accuracy</td>
<td>17-10</td>
</tr>
<tr>
<td>Device-Connect-Time Measurement</td>
<td>17-10</td>
</tr>
<tr>
<td>Device-Connect-Time-Measurement Mode</td>
<td>17-10</td>
</tr>
<tr>
<td>Device-Connect-Time-Measurement Enable</td>
<td>17-11</td>
</tr>
<tr>
<td>Extended Measurement Word</td>
<td>17-11</td>
</tr>
<tr>
<td>Extended-Measurement-Word Enable</td>
<td>17-11</td>
</tr>
<tr>
<td>Signals and Resets</td>
<td>17-12</td>
</tr>
<tr>
<td>Signals</td>
<td>17-12</td>
</tr>
<tr>
<td>Halt Signal</td>
<td>17-12</td>
</tr>
<tr>
<td>Clear Signal</td>
<td>17-12</td>
</tr>
<tr>
<td>Reset Signal</td>
<td>17-13</td>
</tr>
<tr>
<td>Resets</td>
<td>17-13</td>
</tr>
<tr>
<td>Channel-Path Reset</td>
<td>17-13</td>
</tr>
<tr>
<td>I/O-System Reset</td>
<td>17-13</td>
</tr>
<tr>
<td>Externally Initiated Functions</td>
<td>17-17</td>
</tr>
<tr>
<td>Status Verification</td>
<td>17-23</td>
</tr>
<tr>
<td>Address-Limit Checking</td>
<td>17-23</td>
</tr>
<tr>
<td>Configuration Alert</td>
<td>17-24</td>
</tr>
<tr>
<td>Incorrect-Length-Indication Suppression</td>
<td>17-24</td>
</tr>
<tr>
<td>Concurrent Sense</td>
<td>17-24</td>
</tr>
<tr>
<td>Channel-Subsystem Recovery</td>
<td>17-24</td>
</tr>
<tr>
<td>Channel Report</td>
<td>17-25</td>
</tr>
<tr>
<td>Channel-Subsystem-I/O-Priority Facility</td>
<td>17-29</td>
</tr>
<tr>
<td>Number of Channel-Subsystem-Priority Levels</td>
<td>17-29</td>
</tr>
<tr>
<td>Signals and Resets</td>
<td>17-12</td>
</tr>
<tr>
<td>Signals</td>
<td>17-12</td>
</tr>
<tr>
<td>Halt Signal</td>
<td>17-12</td>
</tr>
<tr>
<td>Clear Signal</td>
<td>17-12</td>
</tr>
<tr>
<td>Reset Signal</td>
<td>17-13</td>
</tr>
<tr>
<td>Resets</td>
<td>17-13</td>
</tr>
<tr>
<td>Channel-Path Reset</td>
<td>17-13</td>
</tr>
<tr>
<td>I/O-System Reset</td>
<td>17-13</td>
</tr>
<tr>
<td>Externally Initiated Functions</td>
<td>17-17</td>
</tr>
<tr>
<td>Status Verification</td>
<td>17-23</td>
</tr>
<tr>
<td>Address-Limit Checking</td>
<td>17-23</td>
</tr>
<tr>
<td>Configuration Alert</td>
<td>17-24</td>
</tr>
<tr>
<td>Incorrect-Length-Indication Suppression</td>
<td>17-24</td>
</tr>
<tr>
<td>Concurrent Sense</td>
<td>17-24</td>
</tr>
<tr>
<td>Channel-Subsystem Recovery</td>
<td>17-24</td>
</tr>
<tr>
<td>Channel Report</td>
<td>17-25</td>
</tr>
<tr>
<td>Channel-Report Word</td>
<td>17-26</td>
</tr>
<tr>
<td>Extended-Subchannel-Logout Facility</td>
<td>17-28</td>
</tr>
<tr>
<td>Channel-Subsystem-I/O-Priority Facility</td>
<td>17-29</td>
</tr>
<tr>
<td>Number of Channel-Subsystem-Priority Levels</td>
<td>17-29</td>
</tr>
</tbody>
</table>

The I/O support functions are those functions of the channel subsystem that are not directly related to the initiation or control of I/O operations. The following I/O support functions are described in this chapter:

- Channel-subsystem monitoring
- Signals and resets
- Externally initiated functions
- Status verification
- Address-limit checking
- Configuration alert
- Incorrect-length-indication suppression
- Concurrent sense
- Channel-subsystem recovery
- I/O-priority facility

Channel-Subsystem Monitoring

Monitoring facilities are provided in the channel subsystem so that the program can retrieve measured values on performance for a designated sub-channel. The use of these facilities is under program control by means of the execution of the SET CHANNEL MONITOR instruction and the MODIFY SUBCHANNEL instruction.

The principal components of the channel-subsystem-monitoring facilities are the channel-subsystem-timing facility, measurement-block-update facility, and device-connect-time-measurement facility. The measurement-block-update facility and device-connect-time-measurement facility both use the channel-subsystem-timing facility but otherwise are logically distinct and operate independent of one another.
Other components of the channel-subsystem-monitoring facilities are the control-unit-queuing-measurement facility, the control-unit-defer-time facility, the device-active-only-measurement facility, the initial-command-response-measurement facility, the extended-I/O-measurement-block facility, and the extended-I/O-measurement-word facility. These enhance the measurements of the measurement-block-update facility if they are available as described in later sections, where each of the facilities that constitute the channel-subsystem-monitoring facilities is described in this chapter.

Channel-Subsystem Timing

The channel-subsystem-timing facility provides the channel subsystem with the capability of measuring the elapsed time required for performing several different phases of the processing of a start function initiated by START SUBCHANNEL. These elapsed-time measurements are used by both the measurement-block-update facility and the device-connect-time-measurement facility to provide subchannel performance information to the program.

While every channel subsystem has a channel-subsystem-timing facility, it may or may not be provided for use with all subchannels. Subchannels for which the facility is provided have the timing-facility bit, bit 14 of word 1, stored as one in the associated subchannel-information block. (See “Timing Facility (T)” on page 15-4) If the channel-subsystem-timing facility is not provided for the subchannel, the subchannel’s timing-facility bit is stored as zero.

Subchannels that do not have the channel-subsystem-timing facility provided are those for which the characteristics of the associated device, the manner in which it is attached to the channel subsystem, or the channel-subsystem resources required to support the device are such that use of the channel-subsystem-timing facility is precluded.

The channel-subsystem-timing facility consists of at least one channel-subsystem timer and the associated logic and storage required for computing and recording the elapsed-time intervals for use by the two measurement facilities. The aspects of the channel-subsystem-timing facility that are of importance to the program are described below.

Channel-Subsystem Timer

Each channel-subsystem timer is a binary counter that is not accessible to the program. The channel-subsystem timer provides a minimum timer resolution of 128 microseconds. A timer resolution of 1.0 microseconds is provided when FICON-I/O-interface channel paths are supported. When incrementing the channel-subsystem timer causes a carry out of the leftmost bit position, the carry is ignored, and counting continues from zero. No indications are generated as a result of the overflow.

Just as every CPU has access to a TOD clock, every channel subsystem has access to at least one channel-subsystem timer. When multiple channel-subsystem timers are provided, synchronization among these timers is also provided, creating the effect that all the timing facilities of the channel subsystem share a single timer. Synchronization among these timers may be supplied either through some TOD clock or independently by the channel subsystem.

If the TOD clocks are not synchronized, the elapsed times measured by the channel-subsystem-timing facility may have unpredictable values for some or all of the subchannels, depending on the model and on the particular channel-subsystem timer and the way the associated devices are physically attached to the system. The values are unpredictable for those devices attached to the system by separately configurable channel paths whose associated CPU TOD clocks are not synchronized.

Synchronization: If either the measurement-block-update mode or the device-connect-time-measurement mode is active and any of the channel-subsystem timers is found to be out of synchronization, a channel-subsystem-timer-sync check is recognized, and a channel report is generated to alert the program (see “Channel-Subsystem Recovery” on page 17-24). If neither of these modes is active, the lack of synchronization is not recognized.
Measurement-Block Update

The measurement-block-update facility provides the program with the capability of accumulating performance information for subchannels that are enabled for the measurement-block-update mode when the measurement-block-update mode is active. A subchannel is enabled for the measurement-block-update mode by setting bit 11 of word 1 of the SCHIB operand to one and then issuing MODIFY SUBCHANNEL. The measurement-block-update mode is made active by the execution of SET CHANNEL MONITOR when bit 62 of general register 1 is one.

When the measurement-block-update mode is active and the subchannel is enabled for the measurement-block-update mode, information is accumulated in a measurement block associated with the subchannel. A measurement block is either a 32-byte area (format-0 measurement block) or a 64-byte area (format-1 measurement block) in main storage that is associated with a subchannel for the purpose of accumulating measurement data.

For format-0 measurement blocks, the program specifies a contiguous area of absolute storage, referred to as the measurement-block area, and subdivides this area into 32-byte blocks, one block for each subchannel for which measurement data is to be accumulated. The measurement-block-update facility uses the measurement-block index contained at the subchannel in conjunction with the measurement-block origin established by the execution of SET CHANNEL MONITOR to compute the absolute address of the measurement block associated with a subchannel.

For format-1 measurement blocks, the program provides a 64-byte contiguous area of absolute storage for the subchannel. The measurement-block-update facility uses the measurement-block address provided by the MSCH instruction to access the measurement block.

Measurement data is stored in the measurement block associated with the subchannel each time an I/O operation or chain of I/O operations initiated by a START SUBCHANNEL instruction or RESUME SUBCHANNEL instruction is suspended or is completed at the device. The completion of an I/O operation or chain of I/O operations at the device is normally determined when secondary status is accepted from the device.

The measurement data accumulated in the format-0 and format-1 measurement blocks by the measurement-block-update facility is described in the following section, “Measurement Block.”

Measurement Block

A measurement block is either a 32-byte area (format-0 measurement block) at a location designated by the program by its use of a measurement-block-origin address in conjunction with the measurement-block index, or a 64-byte area (format-1 measurement block) at a location designated by the measurement-block address provided in the SCHIB during the execution of the MODIFY SUBCHANNEL instruction. The measurement block contains the accumulated values of the measurement data described below. When the measurement-block-update mode is active and the subchannel is enabled for measurement-block update, the measurement-block-update facility accumulates the values for the measurement data that accrue during the performance of an I/O operation or chain of I/O operations initiated by START SUBCHANNEL.

When the I/O operation or chain of I/O operations is suspended or completed and no error condition is encountered, the accrued values are added to the accumulated values in the measurement block for that subchannel. If an error condition is detected and subchannel-logout information is stored in the extended-status word (ESW), the accrued values are not added to the accumulated values in the measurement block for the subchannel, and the two count fields in the measurement block are not incremented.

If (1) any of the accrued time values is detected to exceed the internal storage provided for containing these values, (2) the control unit cannot provide an accurate queuing time or defer time for the current operation, or (3) the channel subsystem successfully recovers from certain error conditions, none of the accrued values is added to the measurement block for the subchannel, and the sample count in the measurement block is not incremented, but the SSCH+RSCH count in the block is incremented.
References to the measurement block by the measurement-block-update facility, in order to accumulate measurement data at the suspension or completion of an I/O function, are single-access references and appear to be word concurrent as observed by CPUs.

The measurement-block-update facility updates all fields in the measurement block that are required to be updated for a suspended I/O operation prior to putting the subchannel into the suspended state. The measurement-block-update facility updates all fields in the measurement block that are required to be updated for a completed I/O operation prior to making the subchannel status pending with secondary status or, if the subchannel is start pending for a subsequent operation, prior to initiating the start function.

A format-0 measurement block is stored when the measurement-block-format-control bit at the subchannel is zero; a format-1 measurement block is stored when the measurement-block-format-control bit at the subchannel is one.

The format-0 measurement block has the following format:

<table>
<thead>
<tr>
<th>Word 0</th>
<th>SSCH+RSCH Count</th>
<th>Sample Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Device-Connect Time</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Function-Pending Time</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Device-Disconnect Time</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Control-Unit-Queuing Time</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Device-Active-Only Time</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Device-Busy Time</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Initial-Command-Response Time</td>
<td></td>
</tr>
</tbody>
</table>

The format-1 measurement block has the following format:

<table>
<thead>
<tr>
<th>Word 0</th>
<th>SSCH+RSCH Count</th>
<th>Sample Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sample Count</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Device-Connect Time</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Function-Pending Time</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Device-Disconnect Time</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Control-Unit-Queuing Time</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Device-Active-Only Time</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Device-Busy Time</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Initial-Command-Response Time</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SSCH+RSCH Count:** Bits 0-15 of word 0 in the format-0 measurement block and bits 0-31 of word 0 in the format-1 measurement block are used as a binary counter. During the performance of a start function for which measurement-block update is active, when (1) the secondary status condition is recognized or (2) the suspend function is performed, the counter is incremented by one, and the measurement data is stored. The counter wraps around from the maximum value to 0. The program is not alerted when counter overflow occurs.

If the measurement-block-update mode is active and the subchannel is enabled for measuring, the SSCH+RSCH count is incremented even when the lack of measured values for an individual start function precludes the updating of the remaining fields of the measurement block or when the timing-facility bit for the subchannel is zero. The SSCH+RSCH count is not incremented if the measurement-block-update mode is inactive, if the subchannel is not enabled for the measurement-block update, or if subchannel-logout information has been generated for the start function.
**Sample Count:** Bits 16-31 of word 0 in the format-0 measurement block and bits 0-31 of word 1 in the format-1 measurement block are used as a binary counter. When the time-accumulation fields following word 0 of the measurement block are updated, the counter is incremented by one. On some models, certain conditions may prevent the measurement-block-update facility from obtaining the accrued values of the measurement data for an individual start function, even when the measurement-block-update mode is active and the subchannel is enabled for that mode. The control unit may also signal that it was not able to accumulate an accurate queuing time. In these situations, the sample-count field is not incremented.

The counter wraps around from the maximum value to 0. The program is not alerted when counter overflow occurs. This field is not updated if the channel-subsystem-timing facility is not provided for the subchannel.

The System Library publication for the system model specifies the conditions, if any, that preclude the updating of the sample count and time-accumulation fields of the measurement block.

**Device-Connect Time:** Bit positions 0-31 of word 1 in the format-0 measurement block and bit positions 0-31 of word 2 in the format-1 measurement block contain the accumulation of measured device-connect-time intervals. The device-connect-time interval (DCTI) is the sum of the time intervals measured whenever the device is logically connected to a channel path while the subchannel is subchannel active and the device is actively communicating with the channel path. The device-connect time does not include the intervals when a device is logically connected to a channel path but is not actively communicating with the channel. The device reports the accumulation of time intervals when the device is logically connected but not actively communicating with the channel path as control-unit-defer time. The control-unit-defer time is not included in the device-connect-time measurement but, instead, is added to the accrued device-disconnect-time measurement for the operation.

The time intervals are stored using a resolution of 128 microseconds. The accumulated value is modulo approximately 152.71 hours, and the program is not alerted when an overflow occurs. This field is not updated if (1) the channel-subsystem-timing facility is not provided for the subchannel, (2) the measurement-block-update mode is inactive, or (3) any of the time values accrued for the current start function has been detected to exceed the internal storage in which it was accrued.

Accumulation of device-connect-time intervals for a subchannel and storing this data in the ESW are not affected by whether the measurement-block-update mode is active. (See "Device-Connect-Time Measurement" on page 17-10.)

**Function-Pending Time:** Bit positions 0-31 of word 2 in the format-0 measurement block and bit positions 0-31 of word 3 in the format-1 measurement block contain the accumulated SSCH- and RSCH-function-pending time. Function-pending time is the time interval between acceptance of the start function (or resume function if the subchannel is in the suspended state) at the subchannel and acceptance of the first command associated with the initiation or resumption of channel-program execution at the device.

When channel-program execution is suspended because of a suspend flag in the first CCW of a channel program, the suspension occurs prior to transferring the first command to the device. In this case, the function-pending time accumulated up to that point is added to the value in the function-pending-time field of the measurement block. Function-pending time is not accrued while the subchannel is suspended. Function-pending time begins to be accrued again, in this case, when RESUME SUBCHANNEL is subsequently executed while the designated subchannel is in the suspended state.

The function-pending-time interval is stored using a resolution of 128 microseconds. The accumulated value is modulo approximately 152.71 hours, and the program is not alerted when an overflow occurs. This field is not updated if the channel-subsystem-timing facility is not provided for the subchannel.

**Device-Disconnect Time:** Bit positions 0-31 of word 3 in the format-0 measurement block and bit positions 0-31 of word 4 in the format-1 measurement block contain the accumulated device disconnect time. Device-disconnect time is the sum of the time intervals measured whenever the device is logically disconnected from the channel.
subsystem while the subchannel is subchannel active. The device-disconnect time also includes the sum of control-unit-defer-time intervals reported by the device during the I/O operation.

Device-disconnect time is not accrued while the subchannel is in the suspended state. Device-disconnect time begins to be accrued again, in this case, on the first device disconnection after channel-program execution has been resumed at the device (the subchannel is again subchannel active).

The device-disconnect-time interval is stored using a resolution of 128 microseconds. The accumulated value is modulo approximately 152.71 hours; the program is not alerted when an overflow occurs. This field is not updated if the channel-subsystem-timing facility is not provided for the subchannel.

The device-disconnect time does not include the interval between the primary status condition and the secondary status condition at the end of an I/O operation when the subchannel is no longer subchannel active but the I/O device is active. If the channel subsystem provides the device-active-only measurement facility, this time is accumulated in the device-active-only-time field of the measurement block.

**Control-Unit-Queuing Time:** Bit positions 0-31 of word 4 in the format-0 measurement block and bit positions 0-31 of word 5 in the format-1 measurement block contain the accumulated control-unit-queuing time. Control-unit-queuing time is the sum of the time intervals measured by the control unit whenever the device is logically disconnected from the channel subsystem during an I/O operation while the device is busy with an operation initiated from a different system.

Control-unit-queuing time is not accrued while the subchannel is in the suspended state. Control-unit-queuing time may be accrued for the channel program after the subchannel becomes subchannel active following a successful resumption.

The control-unit-queuing-time field is updated such that bit 31 represents 128 microseconds. The accumulated value is modulo approximately 152.71 hours; the program is not alerted when an overflow occurs. This field is not updated if the channel-subsystem-timing facility is not provided for the subchannel or if the control unit does not provide a queuing time.

**Device-Active-Only Time:** Bit positions 0-31 of word 5 in the format-0 measurement block and bit positions 0-31 of word 6 in the format-1 measurement block contain the accumulated device-active-only time. Device-active-only time is the sum of the time intervals when the subchannel is device active but not subchannel active at the end of an I/O operation or chain of I/O operations initiated by a start function or resume function.

Device-active-only time is not accumulated when the subchannel is device active during periods when the subchannel is active; such time is accumulated as device-connect time or device-disconnect time, as appropriate.

The device-active-only-time field is updated such that bit 31 represents 128 microseconds. The accumulated value is modulo approximately 152.71 hours; the program is not alerted when an overflow occurs. This field is not updated if the channel-subsystem-timing facility is not provided for the subchannel.

**Device-Busy Time:** When the extended-I/O-measurement-block facility is installed, bit positions 0-31 of word 6 in the format-0 measurement block, and bit positions 0-31 of word 7 in the format-1 measurement block contain the accumulated device-busy time. Device-busy time is the sum of the time intervals when the subchannel is device busy during an attempt to initiate a start function or resume function at the subchannel.

The device-busy-time field is updated such that bit 31 represents 128 microseconds. The accumulated value is modulo approximately 152.71 hours; the program is not alerted when an overflow occurs. This field is not updated if the channel-subsystem-timing facility is not provided for the subchannel.

**Initial-Command-Response Time:** Bit positions 0-31 of word 7 in the format-0 measurement block, and bit positions 0-31 of word 8 in the format-1 measurement block contain the accumulated initial-command-response time for the subchannel. The initial-command-response time for a start or resume function is the time interval beginning from when the first command of the channel
program is sent to the device until the device indicates it has accepted the command.

The initial-command-response time is stored at a resolution of 128 microseconds. The accumulated value is modulo approximately 152.71 hours; the program is not alerted when an overflow occurs.

This field is not updated if the channel-subsystem-timing facility is not provided for the subchannel or if the initial-command-response-measurement facility is not installed.

**Control-Unit-Defer Time:** Control-unit-defer time is the sum of the time intervals measured by the control unit whenever the device is logically connected to the channel subsystem during an I/O operation but is not actively communicating with the channel because of device-dependent delays in channel-program execution. The control-unit-defer time is not stored in the measurement block as a separate measurement field but is used in the calculation of device-connect-time measurement and device-disconnect-time measurement for an operation.

Control-unit-defer time, if provided by a control unit, is accrued while the device is logically connected to the channel. The time is reported to the channel when channel-end status is presented that causes a device disconnection or terminates the I/O operation. Control-unit-defer time is subtracted from the device-connect-time measurement and is added to the device-disconnect-time measurement reported for the operation.

**Reserved:** The remaining words of the measurement block, along with any words associated with facilities that are not provided by the channel subsystem or the subchannel, are reserved for future use. They are not updated by the measurement-block-update facility.

**Programming Note:** On models that do not have the z/Architecture installed, it is possible for the program to fetch a portion of a measurement block immediately prior to a measurement block update, for the measurement-block-update facility to perform a block-concurrent update of the block, and for the program to then fetch the remainder of the now-updated measurement block. To ensure that a consistent measurement block is fetched, the program should re-fetch the sample-count field after fetching the entire measurement block and compare the re-fetched count with the sample-count value originally fetched. If the two values match, a consistent measurement block has been obtained. If they do not match, the entire measurement block should be re-fetched in the same manner.

On models that have the z/Architecture installed, it is possible for the program to fetch a portion of a measurement block immediately prior to, or during, a measurement block update by the measurement-block-update facility. To ensure that a consistent measurement block is fetched, the program should not fetch measurement data for a subchannel during the time from when a start or resume function is initiated at the subchannel until the operation is suspended or completes with secondary status.

**Measurement-Block Format**

The measurement block is stored as either a format-0 measurement block or a format-1 measurement block. The format-0 measurement block is a 32-byte area (format-0 measurement block) at a location designated by the program using the measurement-block origin in conjunction with the measurement-block index. The format-1 measurement block is a 64-byte area (format-1 measurement block) at a location designated by the measurement-block address provided in the SCHIB during the execution of the MODIFY SUB-CHANNEL instruction. The measurement-block-format-control bit at the subchannel indicates whether a format-0 or format-1 measurement block is stored when measurement-block-update mode is active and enabled at the subchannel.

**Measurement-Block Origin**

The measurement-block origin is the beginning of the measurement-block area in main storage, used to store format-0 measurement blocks. The absolute address of the measurement-block origin, specified on a 32-byte boundary, is passed in general register 2 to the measurement-block-update facility when SET CHANNEL MONITOR is executed with bit 62 of general register 1 set to one.
Measurement-Block Address
The measurement-block address is set at the subchannel through the execution of MODIFY SUB-CHANNEL. The measurement block address specifies the absolute address of the beginning of the 64-byte area to be used for accumulating format-1 measurement-block parameters for that subchannel.

Programming Note: The initial value of the measurement-block address is zero. The program is responsible for setting the measurement-block address to the proper value prior to enabling the subchannel for the measurement-block-update mode and making the mode active.

Measurement-Block Key
Bits 32-35 of general register 1 form the four-bit access key to be used for subsequent measurement-block updates when SET CHANNEL MONITOR causes the measurement-block-update mode to be made active. The measurement-block key is passed to the measurement-block-update facility whenever the measurement-block-update mode is made active. The key is used for both format-0 and format-1 measurement block updates.

Measurement-Block Index
The measurement-block index is set in the subchannel through the execution of MODIFY SUB-CHANNEL. The measurement-block index specifies which 32-byte measurement block, relative to the measurement-block origin, is to be used for accumulating the format-0 measurement-block parameters for that subchannel. The location of the format-0 measurement block of a subchannel is computed by the measurement-block-update facility by appending five rightmost zeros to the measurement-block index of the subchannel and adding the result to the measurement-block origin. The result is the absolute address of the 32-byte format-0 measurement block for that subchannel. When the computed measurement-block address exceeds 2^{31}-1, a measurement-block program-check condition is recognized, and measurement-block updating does not occur for the preceding subchannel-active period.

Programming Note: The initial value of the measurement-block index is zero. The program is responsible for setting the measurement-block index to the proper value prior to enabling the subchannel for the measurement-block-update mode and making the mode active. To ensure predictable results for the measured parameters in the measurement block, each subchannel for which measured parameters are to be accumulated must have a different value for its measurement-block index.

Measurement-Block-Update Mode
The measurement-block-update mode is made active by the execution of SET CHANNEL MONITOR with bit 62 of general register 1 set to one. If bit 62 of general register 1 is zero when SET CHANNEL MONITOR is executed, the mode is made inactive. When the measurement-block-update mode is inactive, no measurement values are accumulated in main storage. When the measurement-block-update mode is made active, the contents of general register 2 are passed to the measurement-block-update facility as the absolute address of the measurement-block origin and is used to calculate the address of format-0 measurement blocks. The measurement-block origin is not used for format-1 measurement blocks. The MBK is also passed to the measurement-block-update facility as the access key to be used when updating either format-0 or format-1 measurement blocks. When the measurement-block-update mode is active, the measurement-block-update facility accumulates measurements in individual measurement blocks for subchannels whose measurement-block-update-enable bit is one. (See the section “Measurement Block” on page 17-3 for a description of the measured parameters.)

If the measurement-block-update mode is already active when SET CHANNEL MONITOR is executed, the values for the measurement-block origin and measurement-block key that are used for a subchannel enabled for measuring by the measurement-block-update facility are dependent upon whether SET CHANNEL MONITOR is executed prior to, during, or subsequent to the execution of START SUBCHANNEL for that subchannel. If SET CHANNEL MONITOR is executed prior to START SUBCHANNEL, the current measurement-block origin and measurement-block key are in control. If SET CHANNEL MONITOR is executed during or subsequent to execution of START SUBCHANNEL, it is unpredictable whether the measurement-block origin and
Measurement-Block Format Control

Bit 29, word 6, of the SCHIB is the measurement-block-format-control bit. This bit provides the capability of specifying whether a format-0 or format-1 measurement block is stored on a sub-channel basis. The initial value of the bit is zero. When MODIFY SUBCHANNEL is executed with the measurement-block-format-control bit in the SCHIB operand set to one, the format-1 measurement block is specified for the subchannel. If the measurement-block-update mode is active and enabled at the subchannel, the measurement-block-update facility stores a format-1 measurement-block for the subchannel, starting with the next START SUBCHANNEL issued to that subchannel. Similarly, if MODIFY SUBCHANNEL is executed with measurement-block-format-control bit of the SCHIB operand set to zero by the program, the measurement-block-update facility stores a format-0 measurement-block for the subchannel, starting with the next START SUBCHANNEL issued to that subchannel.

Measurement-Block-Update Enable

Bit 11 of word 1 of the SCHIB is the measurement-block-update-enable bit. This bit provides the capability of controlling the accumulation of measurement-block parameters on a sub-channel basis. The initial value of the enable bit is zero. When MODIFY SUBCHANNEL is executed with the enable bit set to one in the SCHIB, the subchannel is enabled for the measurement-block-update mode. If the measurement-block-update mode is active, the measurement-block-update facility accumulates measurement-block parameters for the subchannel, starting with the next START SUBCHANNEL issued to that subchannel. Conversely, if MODIFY SUBCHANNEL is executed with bit 11 of word 1 of the SCHIB operand set to zero by the program, the subchannel is disabled for the measurement-block-update mode, and no additional measurement-block parameters are accumulated for that subchannel.

Control-Unit-Queuing Measurement

The control-unit-queuing-measurement facility allows the channel subsystem to accept queuing times from control units and, in conjunction with the measurement-block-update facility, to accumulate those times in the measurement block.

The System Library publication for the control-unit model specifies its ability to supply queuing time. If a control-unit model is capable of supplying queuing time, the publication specifies the conditions that prevent the control unit from accumulating an accurate control-unit-queuing time.

Control-Unit-Defer Time

The control-unit-defer-time facility allows the channel subsystem to accept defer times from control units and, in conjunction with the measurement-block-update facility, to modify the device-connect and device-disconnect times reported in the measurement block to reflect the defer time. The control-unit-defer time is subtracted from the device-connect-time measurement and is added to the device-disconnect-time measurement reported for an I/O operation.

The System Library publication for the control-unit model specifies its ability to supply defer time. If a control-unit model is capable of supplying defer time, the publication specifies the conditions that prevent the control unit from accumulating an accurate control-unit-defer time.

Device-Active-Only Measurement

The device-active-only-measurement facility permits the channel subsystem to report the times that the device is disconnected between primary status and secondary status at the end of an I/O operation or chain of I/O operations.

The measurement-block updates are performed at the time that secondary status is accepted from the I/O device, in order that the device-active time between primary status and secondary status can be reported.

If the subchannel is start pending when secondary status is accepted from the I/O device and the measurement-block update is to be performed, the measurement-block update is performed prior to performing the start function. If measurement-block errors occur, they are reported to the
program along with the secondary status instead of performing the start function.

Initial-Command-Response Measurement

The initial-command-response-measurement facility allows the channel subsystem to calculate initial-command-response time for I/O operations and, in conjunction with the measurement-block-update facility, to accumulate those times in the measurement block. The initial-command-response time is accumulated in word 7 of the measurement block.

Time-Interval-Measurement Accuracy

On some models, when time intervals are to be measured and condition code 0 is set for START SUBCHANNEL (or RESUME SUBCHANNEL in the case of a suspended subchannel), a period of latency may occur prior to the initiation of the function-pending time measurement. The System Library publication for the system model specifies the mean latency value and variance for each of the measured time intervals.

Programming Notes:

1. Excessive delays may be encountered by the channel subsystem when attempting to update measurement data if the program is concurrently accessing the same measurement-block area. A programming convention should ensure that the storage block designated by SET CHANNEL MONITOR is made read-only while the measurement-block-update mode is active.

2. To ensure that programs written to support measurement functions are executed properly, the program should initialize all the measurement blocks to zeros prior to making the measurement-block-update mode active. Only zeros should appear in the reserved and unused words of the measurement blocks.

3. When the incrementing of an accumulated value causes a carry to be propagated out of bit position 0, the carry is ignored, and accumulating continues from zero on.

Device-Connect-Time Measurement

The device-connect-time-measurement facility provides the program with the capability of retrieving the length of time that a device is actively communicating with the channel subsystem while executing a channel program. The measured length of time that the device spends actively communicating on a channel path during the execution of a channel program is called the device-connect-time interval (DCTI). Control-unit-defer time is not included in the DCTI.

If timing facilities are provided for the subchannel, the DCTI value is passed to the program in the extended-status word (ESW) at the completion of the operation when the primary-status condition is cleared by TEST SUBCHANNEL and when TEST SUBCHANNEL clears an intermediate-status condition alone while the subchannel is suspended. The DCTI value passed in the ESW pertains to the previous subchannel-active period. The passing of the DCTI in the ESW is under program control by the SET CHANNEL MONITOR device-connect-time-measurement mode-control bit and the corresponding enable bit in the subchannel. However, the DCTI value is not stored in the ESW if the I/O function initiated by START SUB_CHANNEL is terminated because of an error condition that is described by subchannel logout. See the section "Extended-Status Format 0" on page 16-32. In this case, the extended-status bit (L) of the SCSW is stored as one, indicating that the ESW contains logout information describing the error condition. See the section "Extended Status Word" on page 16-32 for the description of the logout information. If the accrued DCTI value exceeded 8.388608 seconds during the previous subchannel-active period, then the maximum value (FFFF hex) is passed in the ESW.

Device-Connect-Time-Measurement Mode

The device-connect-time-measurement mode is made active by the execution of SET CHANNEL MONITOR when bit 63 of general register 1 is one. If bit 63 of general register 1 is zero when SET CHANNEL MONITOR is executed, the mode is made inactive, and DCTIs are not passed to the program. When timing facilities are provided for the subchannel, the device-connect-time-measurement mode is active, and...
the subchannel is enabled for the mode, the DCTI value is passed to the program in the ESW stored when TEST SUBCHANNEL (1) clears the primary-interruption condition with no logout information indicated in the SCSW (extended-status-word-format bit is zero) or (2) clears the intermediate-status condition alone while the subchannel is suspended.

If a start function is currently being executed with a subchannel enabled for the device-connect-time-measurement mode when SET CHANNEL MONITOR makes this mode active for the channel subsystem, the value of the DCTI stored under the appropriate conditions may be zero, a partial result, or the full and correct value, depending on the model and the progress of the start function at the time the mode was activated.

Provision of the DCTI value in the measurement-block area is not affected by whether the device-connect-time-measurement mode is active.

**Device-Connect-Time-Measurement Enable**

Bit 12 of word 1 of the SCHIB is the device-connect-time-measurement-mode-enable bit. This bit provides the program with the capability of selectively controlling the storing of DCTI values for a subchannel when the device-connect-time-measurement mode is active. The initial value of the enable bit is zero. When this enable bit is one in the SCHIB and MODIFY SUBCHANNEL is executed, the subchannel is enabled for the device-connect-time-measurement mode. If the device-connect-time-measurement mode is active, the device-connect-time-measurement facility begins providing DCTI values for the subchannel, starting with the next START SUBCHANNEL issued to the subchannel. In this situation, the DCTI values are provided in the ESW (see the section "Extended-Status Format 2" on page 16-39). Similarly, if MODIFY SUBCHANNEL is executed with bit 12 of word 1 of the SCHIB operand set to zero by the program, the subchannel is disabled for the device-connect-time-measurement mode, and no further DCTI values are passed to the program for that subchannel.

**Extended Measurement Word**

The extended-I/O-measurement-word facility provides the program with the capability of retrieving measurement information for a channel program. The measurement information is stored into the extended-measurement-word (EMW) in the Interruption Response Block when the extended-measurement-word enable bit is one at the subchannel. See the section "Extended-Measurement Word" on page 16-41 in Chapter 16, "I/O Interruptions," for the description of the extended-measurement word.

When the extended-measurement-word is enabled for the subchannel, measurement values are passed to program in the EMW when TEST SUBCHANNEL clears a primary-status condition, secondary-status condition alone, or an intermediate-status condition alone while the subchannel is suspended. The measurement values stored in the EMW pertain to the previous subchannel-active and device-active period. Measurement values are not stored in the EMW if the I/O function initiated by START SUBCHANNEL is terminated because of an error condition that is described by subchannel logout (see the section "Extended-Status Format 0" on page 16-32). In this case, the extended-status bit (L) of the SCSW is stored as one, indicating that the ESW contains logout information describing the error condition. See the section "Extended-Status Word" on page 16-32 for the description of the logout information. If any of the accrued measurement values exceeded the maximum value capable of being measured during the previous subchannel-active and device-active period, then the maximum value is stored for that value in the EMW.

**Extended-Measurement-Word Enable**

Bit 30 of word 6 of the SCHIB is the extended-measurement-word enable bit. This bit provides the program with the capability of selectively controlling the storing of measurement values for a subchannel. The initial value of the enable bit is zero. When this enable bit is one in the SCHIB and MODIFY SUBCHANNEL is executed, the subchannel is enabled for the extended-measurement-word and the extended-measurement-word facility begins providing measurement values for the subchannel starting with the next START SUBCHANNEL issued to the sub-
Similarly, if MODIFY SUBCHANNEL is executed with bit 30, word 6, of the SCHIB operand set to zero by the program, the subchannel is disabled for the extended-measurement-word and no further measurement values are passed to the program for that subchannel.

Signals and Resets
During system operation, it may become necessary to terminate an I/O operation or to reset either the I/O system or a portion of the I/O system. (The I/O system consists of the channel subsystem plus all of the attached control units and devices.) Various signals and resets are provided for this purpose. Three signals are provided for the channel subsystem to notify an I/O device to terminate an operation or perform a reset function or both. Two resets are provided to cause the channel subsystem to reinitialize certain information contained either at the I/O device or at the channel subsystem.

Signals
The request that the channel subsystem initiate a signaling sequence is made by one of the following:

1. The program's issuance of the CLEAR SUBCHANNEL, HALT SUBCHANNEL, or RESET CHANNEL PATH instruction
2. The I/O device's signaling of I/O-error alert
3. The channel subsystem itself, upon detecting certain error conditions or equipment malfunctions

The three signals are the halt signal, the clear signal, and the reset signal.

Halt Signal
The halt signal is provided so the channel subsystem can terminate an I/O operation. The halt signal is issued by the channel subsystem as part of the halt function performed subsequent to the execution of HALT SUBCHANNEL. The halt signal is also issued by the channel subsystem when certain error conditions are encountered.

For the parallel-I/O-interface type of channel path, the halt signal results in the channel subsystem using the interface-disconnect sequence control defined in the System Library publication IBM System/360 and System/370 I/O Interface Channel to Control Unit OEMI, GA22-6974.

For the ESCON-I/O-interface type of channel path, the halt signal results in the channel subsystem using the cancel function defined in the System Library publication IBM Enterprise Systems Architecture/390 ESCON I/O Interface, SA22-7202.

For the FICON-I/O-interface type of channel path, the halt signal results in the channel subsystem using the cancel function defined in the ANSI standards document Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2).

Clear Signal
The clear signal is provided so the channel subsystem can terminate an I/O operation and reset status and control information contained at the device. The clear signal is issued as part of the clear function performed subsequent to the execution of CLEAR SUBCHANNEL. The clear signal is also issued by the channel subsystem when certain error conditions or equipment malfunctions are detected by the I/O device or the channel subsystem.

For the parallel-I/O-interface type of channel path, the clear signal results in the channel subsystem using the selective-reset sequence control defined in the System Library publication IBM System/360 and System/370 I/O Interface Channel to Control Unit OEMI, GA22-6974.

For the ESCON-I/O-interface type of channel path, the clear signal results in the channel subsystem using the selective-reset function defined in the System Library publication IBM Enterprise Systems Architecture/390 ESCON I/O Interface, SA22-7202.

For the FICON-I/O-interface type of channel path, the clear signal results in the channel subsystem using the selective-reset function defined in the ANSI standards document Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2).

If an I/O operation is in progress at the device and the device is actively communicating over a channel path in the performance of that I/O operation when a clear signal is received on that channel path, the device disconnects from that
channel path upon receiving the clear signal. Data transfer and any operation using the facilities of the control unit are immediately concluded, and the I/O device is not necessarily positioned at the beginning of a block. Mechanical motion not involving the use of the control unit, such as rewinding magnetic tape or positioning a disk-access mechanism, proceeds to the normal stopping point, if possible. The device may appear busy until termination of the mechanical motion or the inherent cycle of operation, if any, whereupon it becomes available. Status information in the device and control unit is reset, but an interruption condition may be generated upon the completion of any mechanical operation.

Reset Signal
The reset signal is provided so the channel subsystem can reset all I/O devices on a channel path. The reset signal is issued by the channel subsystem as part of the channel-path-reset function performed subsequent to the execution of RESET CHANNEL PATH. The reset signal is also issued by the channel subsystem as part of the I/O-system-reset function.

For the parallel-I/O-interface type of channel path, the reset signal results in the channel subsystem using the system-reset sequence control defined in the System Library publication *IBM System/360 and System/370 I/O Interface Channel to Control Unit OEMI*, GA22-6974.

For the ESCON-I/O-interface type of channel path, the reset signal results in the channel subsystem using the system-reset function defined in the System Library publication *IBM Enterprise Systems Architecture/390 ESCON I/O Interface*, SA22-7202.

For the FICON-I/O-interface type of channel path, the reset signal results in the channel subsystem using the system-reset function defined in the ANSI standards document *Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2).*

Resets
Two resets are provided so the channel subsystem can reinitialize certain information contained at either the I/O device or the channel subsystem. The request that the channel subsystem initiate one of the reset functions is made by one of the following:

1. The program's issuance of the RESET CHANNEL PATH instruction
2. The operator's activation of a system-reset-clear or system-reset-normal key or a load-clear or load-normal key
3. The channel subsystem itself upon detecting certain error conditions or equipment malfunctions

The resets are channel-path reset and I/O-system reset.

Channel-Path Reset
The channel-path-reset facility provides a mechanism to reset certain indications that pertain to a designated channel path at all associated subchannels. Channel-path reset occurs when the channel subsystem performs the channel-path-reset function initiated by RESET CHANNEL PATH. (See "RESET CHANNEL PATH" on page 14-9). All internal indications of dedicated allegiance, control unit busy, and device busy that pertain to the designated channel path are cleared in all subchannels, and reset is signaled on that channel path. The receipt of the reset signal by control units attached to that channel path causes all operations in progress and all status, mode settings, and allegiance, pertaining to that channel path, of the control unit and its attached devices to be reset. (See also the description of the system-reset-signal actions in "I/O-System Reset").

The results of the channel-path-reset function on the designated channel path are communicated to the program by means of a subsequent machine-check-interruption condition generated by the channel subsystem (see "Channel-Subsystem Recovery" on page 17-24).

I/O-System Reset
The I/O-system-reset function is performed when the channel subsystem is powered on, when initial program loading is initiated manually (see "Initial Program Loading" on page 17-17), and when the system-reset-clear or system-reset-normal key is activated. The I/O-system-reset function cannot be initiated under program control; it must be initiated manually. I/O-system reset may fail to complete due to malfunctions detected at the channel subsystem or on a channel path. I/O-system reset is performed as part of subsystem reset, which also resets all floating interruption requests,
including pending I/O interruptions. (See “Sub-system Reset” on page 4-52.) Detailed descriptions of the effects of I/O-system reset on the various components of the I/O system appear later in this chapter.

I/O-system reset provides a means for placing the channel subsystem and its attached I/O devices in the initialized state. I/O-system reset affects only the channel-subsystem configuration in which it is performed, including all channel-subsystem components configured to that channel subsystem. I/O-system reset has no effect on any system components that are not part of the channel-subsystem configuration that is being reset. The effects of I/O-system reset on the configured components of the channel subsystem are described in the following sections.

**Channel-Subsystem State:** I/O-system reset causes the channel subsystem to be placed in the initialized state, with all the channel-subsystem components in the states described in the following sections. All operations in progress are terminated and reset, and all indications of prior conditions are reset. These indications include status information, interruption conditions (but not pending interruptions), dedicated-allegiance conditions, pending channel reports, and all internal information regarding prior conditions and operations. In the initialized state, the channel subsystem has no activity in progress and is ready to perform the initial-program-loading (IPL) function or respond to I/O instructions, as described in Chapter 14, “I/O Instructions.”

**Control Units and Devices:** I/O-system reset causes a reset signal to be sent on all configured channel paths, including those which are not physically available (as indicated by the PAM bit being zero) because of a permanent error condition detected earlier. When the reset signal is received by a control unit, control-unit functions in progress, control-unit status, control-unit allegiance, and control-unit modes for the resetting channel path are reset. Device operations in progress, device status, device allegiance, and the device mode for the resetting channel path are also reset. Control-unit and device mode, allegiance, status, and I/O functions in progress for other channel paths are not affected.

For devices that are operating in the single-path mode, an operation can be in progress for, at most, one channel path. Therefore, if the reset signal is received on that channel path, the operation in progress is reset. Devices that have the dynamic-reconnection feature and are operating in the multipath mode, however, have the capability to establish an allegiance to a group of channel paths during an I/O operation, where all the channel paths of the path group are configured to the same channel subsystem. If an operation is in progress for a device that is operating in the multipath mode and the reset signal is received on one of the channel paths of that path group, then the operation in progress is reset for the resetting channel path only. Although the operation in progress cannot continue on the resetting channel path, it can continue on the other channel paths of the path group, subject to the following restrictions:

1. If the device is actively communicating with the channel subsystem on a channel path when it receives the reset signal on that channel path, then the operation is reset unconditionally, regardless of path groups.
2. If the operation is in progress in the multipath mode but the path group consists only of the resetting path, then the operation is reset.
3. Except as noted in item 2, if the operation in progress is currently in a disconnected state (device not actively communicating with the channel subsystem) or is active on another channel path of a path group, system reset has no effect upon the continued performance of the operation.

A control unit is completely reset after the reset signal has been received on all its channel paths, provided no new activity is initiated at the control unit between the receipt of the first and last reset signal. “Completely reset” means that the current operation, if any, at the control unit is terminated and that control-unit allegiance, control-unit status, and the control-unit mode, if any, are reset.

An I/O device is completely reset after the reset signal has been received on all channel paths of all control units by which the device is accessible, provided no new activity is initiated at the device between the receipt of the first and last reset signal. “Completely reset” means that the current operation, if any, at the device is terminated and that device allegiance, device status, and the device mode are reset.
In summary, system reset always causes an operation in progress to be reset for the channel path on which the reset signal is received. If the resetting channel path is the only channel path for which the operation is in progress, then the operation is completely reset. If a device is actively communicating on a channel path over which the reset signal is received, then the operation in progress is unconditionally and completely reset.

The reset signal is not received by control units and devices on channel paths from which the control unit has been partitioned. A control unit is partitioned from a channel path by means of an enable/disable switch on the control unit for each channel path by which it is accessible. Multi-tagged, unsolicited status, if any, remains pending at the control unit for such a channel path in this case. However, from the point of view of the program, the control unit and device appear to be completely reset if the reset signal is received by the control unit on all the channel paths by which it is currently accessible.

The resultant reset state of individual control units and devices is described in the System Library publication for the control unit.

**Channel Paths:** I/O-system reset causes a reset signal to be sent on all configured channel paths and causes the channel subsystem to be placed in the reset and initialized state, as described in the previous sections. As a result of these actions, all communication between the channel subsystem and its attached control units and devices is terminated and the components reset, and all configured channel paths are made quiescent or are deconfigured.

**Subchannels:** I/O-system reset causes all operations on all subchannels to be concluded. Status information, all interruption conditions (but not pending interruptions), dedicated-allegiance conditions, and internal indications regarding prior conditions and operations in all subchannels are reset, and all valid subchannels are placed in the initialized state.

In the initialized state, the subchannel parameters of all valid subchannels are set to their initial values. The initial values of the following subchannel parameters are zeros:

- Interruption parameter
- I/O-interruption-subclass code (ISC)
- Enabled
- Limit mode
- Measurement mode
- Multipath mode
- Path-not-operational mask
- Last-path-used mask
- Measurement-block index
- Concurrent sense

The initial values of the following subchannel parameters are assigned as part of the installation procedure for the device associated with each valid subchannel:

- Timing facility
- Device number
- Logical-path mask (same value as path-installed mask)
- Path-installed mask
- Path-available mask
- Channel-path ID 0-7

The values assigned may depend upon the particular system model and the configuration; dependencies, if any, are described in the System Library publication for the system model. Programming considerations may further constrain the values assigned.

The initial value of the path-operational mask is all ones.

The device-number-valid bit is one for all subchannels having an assigned I/O device.

The initial value of the model-dependent area of the subchannel-information block is described in the System Library publication for the system model.

The initial value of the subchannel-status word and extended-status word is all zeros.

The initialized state of the subchannel is the state specified by the initial values for the subchannel parameters described above. The description of the subchannel parameters can be found in "Subchannel-Information Block" on page 15-1, "Subchannel-Status Word" on page 16-6, and "Extended-Status Word" on page 16-32.

**Channel-Path-Reset Facility:** I/O-system reset causes the channel-path-reset facility to be reset. A channel-path-reset function initiated by RESET CHANNEL PATH, either pending or in progress, is
overridden by I/O-system reset. The machine-check-interruption condition, which normally signals the completion of a channel-path-reset function, is not generated for a channel-path-reset function that is pending or in progress at the time I/O-system reset occurs.

**Address-Limit-Checking Facility:** I/O-system reset causes the address-limit-checking facility to be reset. The address-limit value is initialized to all zeros and validated.

**Channel-Subsystem-Monitoring Facilities:** I/O-system reset causes the channel-subsystem-monitoring facilities to be reset. The measurement-block-update mode and the device-connect-time-measurement mode, if active, are made inactive. The measurement-block origin and the measurement-block key are both initialized to zeros and validated.

**Pending Channel Reports:** I/O-system reset causes pending channel reports to be reset.

**Channel-Subsystem Timer:** I/O-system reset does not necessarily affect the contents of the channel-subsystem timer. In models that provide channel-subsystem-timer checking, I/O-system reset may cause the channel-subsystem timer to be validated.

**Pending I/O Interruptions:** I/O-system reset does not affect pending I/O interruptions. However, during subsystem reset, I/O interruptions are cleared concurrently with the performance of I/O-system reset. (See “Subsystem Reset” on page 4-52.)
<table>
<thead>
<tr>
<th>Area Affected</th>
<th>Effect of I/O-System Reset&lt;sup&gt;1&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel-subsystem state</td>
<td>Reset and initialized</td>
</tr>
<tr>
<td>Control units and devices</td>
<td>Reset</td>
</tr>
<tr>
<td>Channel paths</td>
<td>Quiescent</td>
</tr>
<tr>
<td>Subchannels</td>
<td>Reset and initialized</td>
</tr>
<tr>
<td>Interruption parameter</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>I/O-interruption-subclass code (ISC)</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Enabled bit</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Address-limit-mode bits</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Timing-facility bit</td>
<td>Installed value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Multipath-mode bit</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Measurement-mode bits</td>
<td>Installed value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Device-number-valid bit</td>
<td>Installed value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Logical-path mask</td>
<td>Equal to path-installed mask value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Path-not-operational mask</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Last-path-used mask</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Path-installed mask</td>
<td>Installed value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Measurement-block index</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Path-operational mask</td>
<td>Ones&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Path-available mask</td>
<td>Installed value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Channel-path ID 0–7</td>
<td>Installed value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Concurrent-sense bit</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Subchannel-status word</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Extended-status word</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Model-dependent area</td>
<td>Model dependent&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Channel-path-reset facility</td>
<td>Reset</td>
</tr>
<tr>
<td>Address-limit-checking facility</td>
<td>Reset and initialized</td>
</tr>
<tr>
<td>Address-limit value</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Channel-subsystem-monitoring facility</td>
<td>Reset and initialized</td>
</tr>
<tr>
<td>Measurement-block-update mode</td>
<td>Inactive&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Device-connect-time-measurement mode</td>
<td>Inactive&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Measurement-block origin</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Measurement-block key</td>
<td>Zeros&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Pending channel-report words</td>
<td>Cleared</td>
</tr>
<tr>
<td>Channel-subsystem timer</td>
<td>Unchanged/validated</td>
</tr>
</tbody>
</table>

Explanation:

1. For a detailed description of the effect of I/O-system reset on each area, see the text.

2. Initialized value.

3. Also subject to model-dependent configuration controls, if any.

---

**Externally Initiated Functions**

I/O-system reset, which is an externally initiated function, is described in "I/O-System Reset" on page 17-13.

---

**Initial Program Loading**

Initial program loading (IPL) provides a manual means for causing a program to be read from a designated device and for initiating the execution of that program.

Some models may provide additional controls and indications relating to IPL; this additional information is specified in the System Library publication for the model.
There are two types of IPL: CCW-type IPL and list-directed IPL. CCW-type IPL is provided by all machine configurations. List-directed IPL may be provided, depending on the model.

**CCW-type IPL**

CCW-type IPL is initiated manually by setting the load-unit-address controls to a four-digit number to designate an input device and by subsequently activating the load-clear or load-normal key.

Activating the load-clear or load-normal key sets the architectural mode to the ESA/390 mode. For ease of reference, the additional elements of the description of ESA/390 IPL are given below.

Activating the load-clear key causes a clear reset to be performed on the configuration.

Activating the load-normal key causes an initial CPU reset to be performed on this CPU, CPU reset to be propagated to all other CPUs in the configuration, and a subsystem reset to be performed on the remainder of the configuration.

In the loading part of the operation, after the resets have been performed, this CPU enters the load state. This CPU does not necessarily enter the stopped state during performance of the reset. The load indicator is on while the CPU is in the load state.

Subsequently, if conditions allow, a read operation is initiated from the designated input device and associated subchannel. The read operation is performed as if a START SUBCHANNEL instruction were executed that designated (1) the subchannel corresponding to the device number specified by the load-unit-address controls and (2) an ORB containing all zeros, except for a byte of all ones in the logical-path-mask field. The ORB parameters are interpreted by the channel subsystem as follows:

- **Interruption parameter**: All zeros
- **Subchannel key**: All zeros
- **Suspend control**: Zero (suspension not allowed)
- **CCW format**: Zero
- **CCW prefetch**: Zero (prefetching not allowed)
- **Initial-status-interruption control**: Zero (no request)
- **Address-limit-checking control**: Zero (no checking)

- **Suppress suspended interruption**: Zero (suppression not allowed)
- **Logical-path mask**: Ones (all channel paths logically available)
- **Incorrect-length-suppression mode**: Zero (ignored because format-0 CCWs are specified)
- **Channel-program address**: Absolute address 0

The first CCW to be executed is not fetched from storage. Instead, the effect is as if an implied format-0 CCW, beginning in absolute location 0 and having the following detailed format, were executed:

```
 Loc. 00000100 00000000 0000000000000000
     0  8  16 31
```

In the illustration above, the CCW specifies a read command with the modifier bits zeros, a data address of 0, a byte count of 24, the chain-command flag one, the suppress-incorrect-length-indication flag one, the chain-data flag zero, the skip flag zero, the program-controlled-interruption (PCI) flag zero, the indirect-data-address (IDA) flag zero, and the suspend flag zero. The CCW fetched, as a result of command chaining, from location 8 or 16, as well as any subsequent CCW in the IPL sequence, is interpreted the same as a CCW in any I/O operation, except that any PCI flags that are specified in the IPL channel program are ignored.

At the time the subchannel is made start pending for the IPL read, it is also enabled, which ensures proper handling of subsequent status from the device by the channel subsystem and facilitates subsequent I/O operations using the IPL device. (Except for the subchannel used by the IPL I/O operation, each subchannel must first be made enabled by MODIFY SUBCHANNEL before it can accept a start function or any status from the device.)

When the IPL subchannel becomes status pending for the last operation of the IPL channel program, no I/O-interruption condition is generated. Instead, the subsystem ID is stored in absolute locations 184-187, zeros are stored in absolute locations 188-191, and the subchannel is
cleared of the pending status as if TEST SUB-CHANNEL had been executed but without storing information usually stored in an IRB. If the subchannel-status field that would normally have been stored is all zeros and the device-status field that would normally have been stored contains only the channel-end indication, with or without the device-end indication, the IPL I/O operation is considered to be completed successfully. If the device-end status for the IPL I/O operation is provided separately after channel-end status, it causes an I/O-interruption condition to be generated. When the IPL I/O operation is completed successfully, a new PSW is loaded from absolute locations 0-7. If the PSW loading is successful and no malfunctions are recognized that preclude the completion of IPL, then the CPU leaves the load state, and the load indicator is turned off. If the rate control is set to the process position, the CPU enters the operating state, and CPU operation proceeds under control of the new PSW. If the rate control is set to the instruction-step position, the CPU enters the stopped state, with the manual indicator on, after the new PSW has been loaded.

If the IPL I/O operation or the PSW loading is not completed successfully, the CPU remains in the load state, and the load indicator remains on.

IPL does not complete when any of the following occurs:

- No subchannel contains a valid device number equal to the IPL device number specified by the load-unit-address controls.
- A malfunction is detected in the CPU, main storage, or channel subsystem that precludes the completion of IPL.
- Unsolicited alert status is presented by the device subsequent to the subchannel becoming start pending for the IPL read and before the IPL subchannel becomes subchannel active. The IPL read operation is not initiated in this case.
- The IPL device appeared not operational on all available channel paths to the device, or there were no available channel paths.
- The IPL device presented a status byte containing indications other than channel end, device end, status modifier, control unit end, control unit busy, device busy, or retry status during the IPL I/O operation. Whenever control unit end, control unit busy, or device busy is presented in the status byte, normal path-management actions are taken.
- A subchannel-status indication other than PCI was generated during the IPL I/O operation.
- The PSW loaded from absolute locations 0-7 has a PSW-format error of the type that is recognized early.

Except in the cases of no corresponding subchannel for the device number entered or a machine malfunction, the subsystem ID of the IPL device is stored in absolute locations 184-187; otherwise, the contents of these locations are unpredictable. In all cases of unsuccessful IPL, the contents of absolute locations 0-7 are unpredictable.

Subsequent to a successful IPL, the subchannel parameters contain the normal values as if an actual START SUBCHANNEL had been executed, designating the ORB as described above.

Programming Notes:

1. The information read and placed at absolute locations 8-15 and 16-23 may be used as CCWs for reading additional information during the IPL I/O operation: the CCW at location 8 may specify reading additional CCWs elsewhere in storage, and the CCW at location 16 may specify the transfer-in-channel command, causing transfer to these CCWs.

2. The status-modifier bit has its normal effect during the IPL I/O operation, causing the channel subsystem to fetch and chain to the CCW whose address is 16 higher than that of the current CCW. This applies also to the initial chaining that occurs after completion of the read operation specified by the implicit CCW.

3. The PSW that is loaded at the completion of the IPL operation may be provided by the first eight bytes of the IPL I/O operation or may be placed at absolute locations 0-7 by a subsequent CCW.

4. Activating the load-normal key implicitly specifies the use of the first 24 bytes of main storage and the eight bytes at absolute locations 184-191. Since the remainder of the IPL program may be placed in any part of storage, it is possible to preserve such areas of storage as may be helpful in debugging or
recovery. The IPL program should not be placed in the low 512 bytes of storage since that area is reserved as described in a programming note under "Compatibility between z/Architecture and ESA/390" on page 1-17.

When the load-clear key is activated, the IPL program starts with a cleared machine in a known state, except that information on external storage remains unchanged.

5. When the PSW at absolute location 0 has bit 14 set to one, the CPU is placed in the wait state after the IPL operation is completed. At that point, the load and manual indicators are off, and the wait indicator is on.

**List-Directed IPL**

List-directed IPL supports the use of I/O devices which are not accessed by CCWs. List-directed IPL may be provided for a logical partition or virtual machine, depending on the model and the amount of memory available to the logical partition or virtual machine. The terms logical partition and virtual machine are not defined in this publication. The term logical partition is defined in the machine manuals. The term virtual machine is defined in the virtual-machine product manuals.

List-directed IPL is initiated by activating the load-clear-list-directed or load-with-dump key. Prior to activating either of these keys, the list-directed-IPL parameters are entered using a model-dependent console. List-directed-IPL parameters include the following information:

- Information which identifies the I/O device from which a program is to be loaded. This I/O device is referred to as the load device.
- Information which identifies a location on a load device from which a program will be loaded.
- Information to be used by the program which will be loaded from the load device.

The list-directed-IPL parameters required by a specific machine and the means by which they are entered are specified in the machine manuals.

After the list-directed-IPL parameters have been entered, the load-clear-list-directed or load-with-dump key for a particular CPU is activated. In the description which follows, the term "this CPU" refers to the CPU in the configuration for which the load-clear-list-directed key was activated.

Activating the load-clear-list-directed key causes a clear reset to be performed on all of the CPUs in the configuration.

Activating the load-with-dump key causes a store-status operation to be performed on this CPU. After the store-status operation, an initial-CPU-reset operation is performed on this CPU, CPU reset is propagated to all other CPUs in the configuration, and a subsystem reset is performed on the remainder of the configuration.

In the loading part of the operation, after the resets have been performed, the operations listed below are performed.

- If the rate control is not in the process position, it is forced into the process position.
- If the list-directed IPL operation was initiated by activating the load-with-dump key, the contents of a model-dependent number of contiguous logical-partition or virtual-machine storage locations starting with absolute location 0 are stored in a model-dependent holding area. The data stored includes the contents of all storage locations into which the program will be loaded from the load device. See Programming Note 4 for additional information.
- The program to be loaded from the load device is loaded into logical-partition or virtual-machine storage at the locations specified on the load device.
- The list-directed-IPL parameters are supplied into two formats. One format is an Extensible Markup Language (XML) document. (See the machine manuals for information regarding the format and structure of the XML document, and see “Extensible Markup Language (XML) 1.0,” W3C, for information regarding XML.) The other format is the IPL-parameter list shown in Figure 17-2 on page 17-21.
- The IPL-parameter list is placed in storage locations of the logical partition or virtual machine which do not contain the program loaded from the load device. The first byte of the XML document immediately follows the last byte of the IPL-parameter list.
- The absolute address of the first byte of the IPL-parameter list is stored in absolute locations 20-23.
The format of the IPL-parameter list is shown in Figure 17-2 on page 17-21.

<table>
<thead>
<tr>
<th>Word</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Length</td>
</tr>
<tr>
<td>1</td>
<td>List-Directed IPL Parameters</td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17-2. IPL-Parameter List

Bits 0-31 of word 0 of the IPL-parameter list contain a length field. The length field contains a 32-bit binary number representing the number of bytes in the IPL-parameter list. Words 1 through n of the IPL-parameter list contain additional list-directed-IPL parameters. See the machine manuals for additional information.

If no errors occur during the operations listed above, a new PSW is loaded from absolute locations 0-7. If the PSW loading is successful and if no malfunctions are recognized which preclude execution of the program which was loaded from the load device, then operation proceeds under control of the new PSW.

If errors occur during the operations listed above, or if the PSW loading is not successful, then this CPU remains in the operating state. See the appropriate machine manuals for additional information.

When list-directed IPL was initiated by activating the load-clear-list-directed key, the state of the machine resources when control is passed to the loaded program is as follows:

- This CPU is in the operating state. The contents of the ALB and TLB for this CPU are unpredictable.
- The PSW for this CPU is set to the contents of absolute storage locations 0-7.
- Main storage locations into which the program was loaded from the load device contain the program which was loaded, except for absolute storage locations 20-23. Absolute storage locations 20-23 contain the absolute address of the IPL-parameter list; storage locations starting with the absolute address indicated by the contents of absolute addresses 20-23 contain the IPL-parameter list, which is immediately followed by the XML document containing the list-directed IPL parameters. The contents of all other storage locations are unpredictable.

- The ACC and F bits of the storage keys associated with all main storage locations are set to zero. The R and C bits of the storage keys associated with the first 4 KB block of storage, the blocks of storage containing the XML document and IPL-parameter list, and the blocks of storage containing the loaded program are set to one. The R and C bits of the storage keys associated with all other blocks of storage are unpredictable.
- The TOD clock is unaffected by initial-program loading.

When list-directed IPL was initiated by activating the load-with-dump key, the state of the machine resources when control is passed to the loaded program is as follows:

- The I/O subsystem has been reset.
- The state of all CPUs in the configuration except for this CPU is the same as it was immediately after the CPUs were reset.
- This CPU is in the operating state. The contents of all registers, the ALB, and the TLB for this CPU are unpredictable.
- The PSW for this CPU is set to the contents of absolute storage locations 0-7.
- The contents of all main storage locations which were not stored in the model-dependent holding area are unchanged from the time when the load-with-dump key was activated.
- The contents of all main storage locations which were stored in the model-dependent holding area are unpredictable except as follows:
  - Main storage locations into which the program was loaded from the load device contain the program which was loaded, except for absolute storage locations 20-23.
Absolute storage locations 20-23 contain the absolute address of the IPL-parameter list; storage locations starting with the absolute address indicated by the contents of absolute addresses 20-23 contain the IPL-parameter list, which is immediately followed by the XML document containing the list-directed IPL parameters.

- The storage keys associated with the blocks of main storage which were stored in the model-dependent holding area are unpredictable. The storage keys associated with all other blocks of main storage are in the same state as they were immediately after the CPUs were reset.
- The TOD clock is unaffected by initial-program loading.

List-directed IPL does not complete if any of the following occurs:

- A malfunction is detected in the CPU, channel subsystem, or main storage which precludes the completion of IPL.
- The list-directed IPL parameters did not contain sufficient information to identify the load device or the program on the load device.
- An I/O device designated by the list-directed-IPL parameters is not in the machine configuration.
- The I/O device designated by the list-directed-IPL parameters is a CCW-type device and is not supported by list-directed IPL.
- The I/O device appeared not operational.
- The amount of storage in the logical partition or virtual machine is insufficient to allow list-directed IPL to complete.
- The PSW to be loaded from locations 0-7 has a PSW-format error of the type that is recognized early.

Programming Notes:

1. The executable portion of the program loaded during list-directed IPL should be located in absolute storage starting at absolute storage locations 8192 and higher. This is necessary in order to avoid use of the architecturally assigned storage locations. See "Assign storage locations" on page 3-58).

2. When the address indicated by storage locations 20-23 is above 16 MB, it is necessary for the machine to be in 31-bit addressing mode in order to access the IPL-parameter list and XML document.

3. When list-directed IPL is initiated by activation of the load-clear-list-directed key, the program loaded from the load device is typically an operating-system loader. In this case, the load device may also contain the operating system to be loaded by the operating-system loader. See the applicable operating-system manuals for additional information.

4. When list-directed IPL is initiated by activation of the load-with-dump key, the contents of storage which are affected by the IPL operation are stored in a model-dependent holding area so that they can be retrieved by the program which was loaded from the load device. The program loaded from the load device is typically an O/S-specific system dump program which stores the retrieved data onto a dump device. This process ensures that the contents of storage affected by the IPL operation are preserved for analysis during subsequent debug operations. See the applicable operating system manuals for additional information.

Reconfiguration of the I/O System

Reconfiguration of the I/O system is handled in a model-dependent manner. For example, changes may be made under program control, by using the model-dependent DIAGNOSE instruction; or manually, by using system-operator configuration controls; or by using a combination of DIAGNOSE and manual controls. The method used depends on the system model. The System Library publication for the system model specifies how the changes are made. The partitioning of channel paths because of reconfiguration is indicated by
the setting of the PAM bits in the SCHIB stored when STORE SUBCHANNEL is executed (see "Path-Available Mask (PAM)" on page 15-7).

Status Verification

The status-verification facility provides the channel subsystem with a means of indicating that a device has presented a device-status byte that has valid CBC but that contained a combination of bits that was inappropriate when the status byte was presented to the channel subsystem. The indication provided to the program in the ESW by the channel subsystem is called device-status check. When the channel subsystem recognizes a device-status-check condition, an interface-control-check condition is also recognized. For a summary of the status combinations considered to be appropriate or inappropriate, see the System Library publications IBM Enterprise Systems Architecture/390 ESCON I/O Interface, SA22-7202, and IBM System/360 and System/370 I/O Interface Channel to Control Unit OEMI, GA22-6974, and the ANSI standards document Fibre Channel - Single-Byte Command Code Sets-2 (FC-SB-2).

Address-Limit Checking

The address-limit-checking facility provides a storage-protection mechanism for I/O data accesses to storage that augments key-controlled protection. When address-limit checking is used, absolute storage is divided into two parts by a program-controlled address-limit value. I/O data accesses can then be optionally restricted to only one of the two parts of absolute storage by the limit mode at each subchannel. The address-limit constraint applies at a higher priority than key-controlled protection, that is, I/O data accesses to the part of main storage that is protected by address-limit checking are prevented even when the subchannel key is zero or matches the key in storage. Address-limit checking does not apply to the fetching of CCWs and IDAWs.

The address-limit-checking facility consists of the following elements:

- The I/O instruction SET ADDRESS LIMIT.
- The limit mode at each subchannel.
- The address-limit-checking-control bit in the ORB.

The execution of SET ADDRESS LIMIT passes the contents of general register 1 to the address-limit-checking facility to be used as the address-limit value. Bits 32 and 48-63 of general register 1 must be zeros to designate a valid absolute address on a 64K-byte boundary; otherwise, an operand exception is recognized, and the execution of the instruction is suppressed.

The limit mode at each subchannel indicates the manner in which address-limit checking is to be performed. The limit mode is set by placing the desired value in bit positions 9 and 10 of word 1 in the SCHIB and executing MODIFY SUBCHANNEL. The settings of these bits in the SCHIB have the following meanings:

00 No limit checking (initialized value).
01 Data address must be equal to or greater than the current address limit.
10 Data address must be less than the current address limit.
11 Reserved. This combination of limit-mode bits causes an operand exception to be recognized when MODIFY SUBCHANNEL is executed.

The address-limit-checking-control bit, bit 11 of word 1 of the ORB, specifies whether address-limit checking is to be used for the start function that is accepted when the execution of START SUBCHANNEL causes the contents of the ORB to be passed to the subchannel. If the address-limit-checking-control bit is zero when the contents of the ORB are passed, address-limit checking is not specified for that start function. If the bit is one, address-limit checking is specified and is under the control of the current address limit and the current setting of the limit mode at the subchannel.

During the performance of the start function, an attempt to access an absolute storage location for data that is protected by an address limit (either high or low) is recognized as an address-limit violation, and the access is not allowed. A program-check condition is recognized, and channel-program execution is terminated, just as when an attempt is made to access an invalid address.
Configuration Alert
The configuration-alert facility provides a detection mechanism for devices that are not associated with a subchannel in the configuration. The configuration-alert facility notifies the program, by means of a channel report, that a device which is not associated with a subchannel has attempted to communicate with the program.

Each device must be assigned to a subchannel during an installation procedure; otherwise, the channel subsystem is unable to generate an I/O-interruption condition for the device. This is because the I/O-interruption code contains the subchannel number that identifies the particular device causing the I/O-interruption condition. When a device that is not associated with a subchannel attempts to communicate with the channel subsystem, the configuration-alert facility generates a channel report in which the unassociated device is identified. For a description of the means by which the program is notified of a pending channel report and how the information in the channel report is retrieved, see "Channel Report" on page 17-25.

Incorrect-Length-Indication Suppression
The incorrect-length-indication-suppression facility allows the indication of incorrect length for immediate operations to be suppressed in the same manner when using format-1 CCWs as when using format-0 CCWs. When the incorrect-length-indication-suppression facility is installed, bit 24 of word 1 of the ORB specifies whether the channel subsystem is to suppress the indication of incorrect length for an immediate operation when format-1 CCWs are used or whether this indication will remain under the control of the SLI flag of the current CCW (as is the case for CCWs not executed as immediate operations). This bit provides the capability for a channel program to operate in the same manner regarding the indication of incorrect length regardless of whether format-0 or format-1 CCWs are used.

Concurrent Sense
The concurrent-sense facility provides a mechanism whereby sense information that is provided by the device can be presented by the channel subsystem to the program in the same IRB that contains the unit-check indication when the subchannel is in the concurrent-sense mode. The concurrent-sense mode is made active at a subchannel for which the concurrent-sense facility is applicable when MODIFY SUBCHANNEL is executed and bit 31 of word 6 of the SCHIB operand is set to one. The concurrent-sense facility is applicable to subchannels that are associated with channel paths by which the channel subsystem can attempt to retrieve sense information from the device without requiring program intervention.

Channel-Subsystem Recovery
The channel subsystem provides various methods for extensive detection of malfunctions and other conditions to ensure the integrity of channel-subsystem operation and to achieve automatic recovery of some malfunctions.

The method used to report a particular malfunction or other condition is dependent upon the severity of the malfunction or other condition and the degree to which the malfunction or other condition can be isolated. A malfunction or other condition in the channel subsystem may be indicated to the program by information being stored by one of the following methods:

1. Information is provided in the IRB describing a condition that has been recognized by either the channel subsystem or device that must be brought to the attention of the program. Generally, this information is made available to the program by the execution of TEST SUBCHANNEL, which is usually executed in response to the occurrence of an I/O interruption. (See "Interruption Action" on page 16-5, for a definition of the information stored, as well as Chapter 6, "Interruptions" on page 6-1.)

2. Information is provided in a channel report describing a machine malfunction affecting the identified facility associated with the channel subsystem. This information is made available to the program by the execution of STORE CHANNEL REPORT WORD, which is
usually executed in response to the occurrence of a machine-check interruption. (See Chapter 11, “Machine-Check Handling” on page 11-1 for a description of the machine-check-interruption mechanism and the contents of the machine-check-interruption code.)

3. Information is provided in a channel report describing a malfunction or other condition affecting a collection of channel-subsystem facilities. This information is made available to the program as indicated in item 2 on page 17-24.

4. Information is provided in the machine-check-interruption code (MCIC) describing a malfunction affecting the continued operational integrity of the channel subsystem. (See “Channel-Subsystem Damage” on page 11-18.)

5. Information is provided in the MCIC describing a malfunction affecting the continued operational integrity of a process or of the system. (See “Instruction-Processing Damage” on page 11-16 and “System Damage” on page 11-16.)

Channel reports are used to report malfunctions or other conditions only when the use of the I/O-interruption facility is not appropriate and in preference to reporting channel-subsystem damage, instruction-processing damage, or system damage.

**Channel Report**

When a malfunction or other condition affecting elements of the channel subsystem has been recognized, a channel report is generated. The performance of recovery actions by the program or by external means may be required to gain recovery from the error condition. The channel report indicates the source of the channel report and the recovery state to the extent necessary for determining the proper recovery action. A channel report consists of one or more channel-report words (CRWs) that have been generated from an analysis of the malfunction or other condition. The inclusion of two or more CRWs within a channel report is indicated by the chaining flag being stored as one in all of the CRWs of the channel report except the last one in the chain.

When a channel report is made pending by the channel subsystem for retrieval and analysis by the program (by means of the execution of STORE CHANNEL REPORT WORD), a malfunction or other condition that affects the normal operation of one or more of the channel-subsystem facilities has been recognized. If the channel report that is made pending is an initial channel report, a machine-check-interruption condition is generated that indicates one or more CRWs are pending at the channel subsystem. A channel report is initial either if it is the first channel report to be generated after the most recent I/O-system reset or if no previously generated reports are pending and the last STORE CHANNEL REPORT WORD instruction that was executed resulted in the setting of condition code 1, indicating that no channel report was pending. When the machine-check interruption occurs and bit 9 of the machine-check-interruption code (channel report pending) is one, a channel report is pending. If the program clears the first CRW of a channel report before the associated machine-check interruption has occurred, some models may reset the machine-check-interruption condition, and the associated machine-check interruption does not occur. A machine-check interruption indicating that a channel report is pending occurs only if the machine-check mask (PSW bit 13) and the channel-report-pending subclass mask, bit 3 of control register 14, are both ones. If the channel report that is made pending is not an initial channel report, a machine-check-interruption condition is not generated. The CRW that is presented to the program in response to the first STORE CHANNEL REPORT WORD instruction that is executed after a machine-check interruption may or may not be part of the initial channel report that caused the machine-check condition to be generated. A pending channel-report word is cleared by any CPU executing STORE CHANNEL REPORT WORD, regardless of whether a machine-check interruption has occurred in any CPU. If a CRW is not pending and STORE CHANNEL REPORT WORD is executed, condition code 1 is set, and zeros are stored at the location designated by the second-opand address. During the execution of STORE CHANNEL REPORT WORD as a result of receiving a machine-check interruption, condition code 1 may be set, and zeros may be stored because (1) the related channel report has been cleared by another CPU or (2) a malfunction occurred during the generation of a channel report. In the latter case, if, during a subsequent
attempt, a valid channel report can be made pending, an additional machine-check-interruption condition is generated.

When a channel report consists of multiple chained CRWs, they are presented to the program in the same order that they are placed in the chain by the channel subsystem as a result of consecutive executions of STORE CHANNEL REPORT WORD. If, for example, the first CRW of a chain is presented to the program as a result of executing STORE CHANNEL REPORT WORD, the CRW that is presented as a result of the next execution of STORE CHANNEL REPORT WORD is the second CRW of the same chain and not a CRW that is part of another channel report.

Channel reports are not presented to the program in any special order, except for channel reports whose first or only CRW indicates the same reporting-source code and the same reporting-source ID. These channel reports are presented to the program in the same order that they are generated by the channel subsystem, but they are not necessarily presented consecutively. For example, suppose the channel subsystem generates channel reports A, B, and C, in that order. The first CRW of channel reports B and C indicates the same reporting-source code and the same reporting-source ID. Channel report B is presented to the program before channel report C is presented, but channel report A may be presented after channel report B and before channel report C.

Programming Notes:

1. The information that is provided in a single CRW may be made obsolete by another CRW that is subsequently generated for the same channel-subsystem facility. Therefore, the information that is provided in one channel report should be interpreted in light of the information provided by all of the channel reports that are pending at a given instant.

2. A machine-check-interruption condition is not always generated when a channel report is made pending. The conditions that result in a machine-check-interruption condition being generated are described earlier in this section.

3. After a machine-check interruption has occurred with bit 9 of the machine-check-interruption code set to one, STORE CHANNEL REPORT WORD should be issued repeatedly until all of the pending channel reports have been cleared and condition code 1 has been set.

4. A CRW-overflow condition can occur if the program does not issue successive STORE CHANNEL REPORT WORD instructions in a timely manner after the machine-check-interruption occurs.

5. The number of CRWs that can be pending at the same time is model dependent. During the existence of an overflow condition, CRWs that would have otherwise been made pending are lost and are never presented to the program.

Channel-Report Word

The channel-report word (CRW) provides information to the program that can be used to facilitate the recovery of an I/O operation, a device, or some element of the channel subsystem, such as a channel path or subchannel.

The format of the CRW is as follows:

```
  0 1 2 3 4 8 10 16 31
S R C RSC A 0 ERC Reporting-Source ID
```

**Solicited CRW (S):** Bit 1, when one, indicates a solicited CRW. A CRW is considered by the channel subsystem to be solicited if it is made pending as the direct result of some action that is taken by the program. When bit 1 is zero, the CRW is unsolicited and has been made pending as the result of an action taken by the channel subsystem that is independent of the program.

**Overflow (R):** Bit 2, when one, indicates that a CRW-overflow condition has been recognized since this CRW became pending and that one or more CRWs have been lost. This bit is one in the CRW that has most recently been set pending when the overflow condition is recognized. When bit 2 is zero, a CRW-overflow condition has not been recognized.

A CRW that is part of a channel report is not made pending, even though the overflow condition does not exist, if an overflow condition prevented
a previous CRW of that report from being made pending.

**Chaining (C):** Bit 3, when one, and when the overflow flag is zero, indicates chaining of associated CRWs. Chaining of CRWs is indicated whenever a malfunction or other condition is described by more than a single CRW. The chaining flag is zero if the channel report is described by a single CRW or if the CRW is the last CRW of a channel report.

The chaining flag is not meaningful if the overflow bit, bit 2, is one.

**Reporting-Source Code (RSC):** Bits 4-7 identify the channel-subsystem facility that is associated with the channel report. Some facilities are further identified in the reporting-source-identification field (see below). The following combinations of bits identify the facilities:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Facility</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0</td>
<td>Monitoring facility</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>Subchannel</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>Channel path</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>Configuration-alert facility</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>Channel subsystem</td>
</tr>
</tbody>
</table>

All other bit combinations in the reporting-source-code field are reserved.

**Ancillary Report (A):** Bit 8, when one, indicates that a malfunction of a system component has occurred that was recognized previously or which has affected the activity of multiple channel-subsystem facilities. When the malfunction affects the activity of multiple channel-subsystem facilities, an ancillary-report condition is recognized for all of the affected facilities except one. This bit, when zero, indicates that this malfunction of a system component was not recognized previously. This bit is meaningful for all channel reports.

Depending on the model, recognition of an ancillary-report condition may not be provided, or it may not be provided for all system malfunctions that affect channel-subsystem facilities. When ancillary-report recognition is not provided, bit 8 is set to zero.

**Error-Recovery Code (ERC):** Bits 10-15, when zero, indicate that the channel subsystem has error information regarding the channel-subsystem facility identified in the reporting-source code, and that the program can now request that information. Otherwise, bit positions 10-15 contain the error-recovery code that defines the recovery state of the channel-subsystem facility identified in the reporting-source code. This field, when used in conjunction with the reporting-source code, can be used by the program to determine whether the identified facility has already been recovered and is available for use or whether recovery actions are still required. The following error-recovery codes are defined:

<table>
<thead>
<tr>
<th>Bits</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>Event-information pending</td>
</tr>
<tr>
<td>0 0 0 0 0 1</td>
<td>Available</td>
</tr>
<tr>
<td>0 0 0 0 1 0</td>
<td>Initialized</td>
</tr>
<tr>
<td>0 0 0 1 1 1</td>
<td>Temporary error</td>
</tr>
<tr>
<td>0 0 1 0 0 0</td>
<td>Installed parameters initialized</td>
</tr>
<tr>
<td>0 0 1 0 1 0</td>
<td>Terminal</td>
</tr>
<tr>
<td>0 0 1 1 0 0</td>
<td>Permanent error with facility not initialized</td>
</tr>
<tr>
<td>0 0 1 1 1 1</td>
<td>Permanent error with facility initialized</td>
</tr>
<tr>
<td>0 1 0 0 0 0</td>
<td>Installed parameters modified</td>
</tr>
</tbody>
</table>

All other bit combinations in the error-recovery-code field are reserved.

The specific meaning of each error-recovery code depends on the particular reporting-source code that accompanies it in a CRW. The error-recovery codes are defined as follows:

**Event-Information Pending:** Event information for the identified facility is available for retrieval by the program. This CRW does not indicate the state of the identified facility.

**Available:** The identified facility is in the same state that the program would expect if the CRW had not been generated.

**Initialized:** The identified facility is in the same state that existed immediately following the I/O-system reset that was part of the most recent system IPL.
**Temporary Error:** The identified facility is not operating in a normal manner or has recognized the occurrence of an abnormal event. It is expected that subsequent actions either will restore the facility to normal operation or will record the appropriate information describing the abnormal event.

**Installed Parameters Initialized:** This state is the same as the initialized state, except that one or more parameters that are associated with the facility and that are not modifiable by the program may have been changed.

**Terminal:** The identified facility is in a state such that an operation that was in progress can neither be completed nor terminated in the normal manner.

**Permanent Error with Facility Not Initialized:** The identified facility is in a state of malfunction, and the channel subsystem has not caused a reset function to be performed for that facility.

**Permanent Error with Facility Initialized:** The identified facility is in a state of malfunction, and the channel subsystem has caused or may have caused a reset function to be performed for that facility.

**Installed Parameters Modified:** One or more parameters of the specified facility have been changed.

**Reporting-Source ID (RSID):** Bit positions 16-31 contain the reporting-source ID, which may, depending upon the condition that caused the channel report and the reporting-source code, either further identify the affected channel-subsystem facility or provide additional information describing the condition that caused the channel report. The RSID field has the following format as a function of the bit settings of the reporting-source code.

<table>
<thead>
<tr>
<th>Reporting-Source Code</th>
<th>Reporting-Source ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 6 7</td>
<td>0000 0000 0000 0000</td>
</tr>
<tr>
<td>0 0 1 1 1</td>
<td>xxxx xxxx xxxx xxxx</td>
</tr>
<tr>
<td>0 1 0 0 0</td>
<td>0000 0000 yyyy yyyy</td>
</tr>
<tr>
<td>1 0 0 1 1</td>
<td>0000 0000 yyyy yyyy</td>
</tr>
</tbody>
</table>

**Extended-Subchannel-Logout Facility**

When the extended-subchannel-logout facility is installed and enabled, all fibre-channel (FC) channel paths can create extended-subchannel logouts, under applicable error conditions, and generate I/O interruptions that indicate such a logout is pending retrieval. The extended-subchannel logouts are stored in and retrieved from a model-dependent number of extended-subchannel-logout buffers associated with each channel path supported by the facility.

When TEST SUBCHANNEL for a subchannel that specifies an FC channel path stores an interruption-response block (IRB) containing a format-0 ESW in which the extended-subchannel-logout-pending (E) bit is 1, an extended-subchannel logout is pending for the specified subchannel. Words 2-3 of the format-0 ESW contain the extended-subchannel-logout descriptor (ESLD) that identifies the logout and the channel path for which the logout is pending.

If all of the buffers for a given channel contain pending extended-subchannel logouts, the logouts will remain pending for a model-dependent time interval. If, during that time interval, a subsequent condition occurs that warrants a new extended-subchannel logout, that logout is not made and is lost. If the time interval has been exceeded and the program has not used the ESLD to retrieve a pending logout, the logout may be removed from the buffer or the buffer may be reused to contain a subsequent logout. When a logout is removed from a buffer and the buffer is not reused, the ESLD is no longer recognized as a valid logout identifier. When a buffer is reused, the ESLD may be recognized as a valid logout identifier albeit for
a logout different from the logout for which the original I/O interruption was generated.

**Channel-Subsystem-I/O-Priority Facility**

The channel-subsystem-I/O-priority facility provides a means by which the program can establish a priority relationship, at the channel subsystem, among the subchannels that are placed into the start-pending state when START SUBCHANNEL is executed and condition code 0 is indicated. For I/O-subchannels that are configured to fibre-channel channel paths (FICON and FICON-converted channel paths), it also provides a means by which the program can establish a priority relationship for I/O operations at the fibre-channel-attached control units.

The program assigns the desired channel-subsystem priority and control-unit priority by specifying the desired priority numbers in the ORB extension when START SUBCHANNEL is executed.

The channel-subsystem-priority number specified in the ORB is used by the channel subsystem to determine the order in which start-pending and resume-pending subchannels are selected when the channel subsystem attempts to initiate a start function or a resume function. See the section "Start Function and Resume Function" on page 15-18 for details about these functions. In general, I/O subchannels that are in the start-pending or resume-pending state and have a higher priority number are selected for start-function or resume-function initiation by the channel subsystem before start-pending or resume-pending subchannels that have a lower priority number. The specific priority selection algorithm used by the channel subsystem for this purpose depends on the model. Additionally, the channel subsystem also applies a fairness selection algorithm in conjunction with the priority selection algorithm when selecting I/O subchannels. The specific fairness selection algorithm also depends on the model. For all models, the channel-subsystem priority and fairness selection algorithms are always applied to I/O subchannels that are either start pending or resume pending. Some models may also apply both algorithms to subchannels that are either clear pending or halt pending. See a model's System-Library publication for a description of the priority and fairness selection algorithms that the model provides and whether these algorithms are also applied to clear-pending or halt-pending subchannels.

The control-unit-priority number specified in the ORB is used by control units attached to fibre-channel channel paths in order to determine the priority of the execution of CCWs at the control unit. See "Control-Unit (CU) Priority:" on page 15-26 for additional information.

**Number of Channel-Subsystem-Priority Levels**

Depending on the model, fewer than 256 channel-subsystem-priority levels may be provided by the channel subsystem. Each priority level that the model provides is designated by an eight-bit unsigned binary integer. The lowest provided channel-subsystem-priority level is designated by the integer 0, and each succeeding higher priority level is designated by the next-higher sequential integer. For example, if the model provides 16 priority levels, they are numbered 0-15, respectively, from the lowest priority level to the highest priority level.
HFP Arithmetic

HFP Number Representation

A hexadecimal-floating-point (HFP) number consists of a sign bit, a hexadecimal fraction, and an unsigned seven-bit binary integer called the characteristic. The characteristic represents a signed exponent and is obtained by adding 64 to the exponent value (excess-64 notation). The range of the characteristic is 0 to 127, which corresponds to an exponent range of −64 to +63. The magnitude of an HFP number is the product of its fraction and the number 16 raised to the power of the exponent that is represented by its characteristic. The number is positive or negative depending on whether the sign bit is zero or one, respectively.

The fraction of an HFP number is treated as a hexadecimal number because it is considered to be multiplied by a number which is a power of 16. The name, fraction, indicates that the radix point is assumed to be immediately to the left of the leftmost fraction digit.

When an HFP operation would cause the result exponent to exceed 63, the characteristic wraps around from 127 to 0, and an HFP-exponent-overflow condition exists. The result characteristic is then too large by 128, except that a zero characteristic is produced when a true zero is forced.

A true zero is an HFP number with a zero characteristic and zero fraction. A true zero may arise as the normal result of an arithmetic operation because of the particular magnitude of the operands. For HFP operations, the result is forced to be a positive true zero when:

1. An HFP exponent underflow occurs and the HFP-exponent-underflow mask bit in the PSW is zero.
2. The result fraction of an addition or subtraction operation is zero and the HFP-significance mask bit in the PSW is zero.
3. The operand of the CONVERT FROM FIXED instruction is zero.
4. The dividend in the DIVIDE instruction has a zero fraction.
5. The operand of the HALVE, LOAD FP INTEGER, or SQUARE ROOT instruction has a zero fraction.
6. One or both operands of a multiplication operation has a zero fraction.

Item 2, above, applies to normalized and unnormalized instructions.

When a program interruption for HFP exponent underflow occurs, a true zero is not forced; instead, the fraction and sign remain correct, and the characteristic is too large by 128. When a
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Nonzero Result Normalized</th>
<th>Zero Result Forced to True Zero</th>
<th>Zero Result Made Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Short and Long</td>
<td>Extended</td>
</tr>
<tr>
<td>ADD NORMALIZED</td>
<td>Yes</td>
<td>Y/N</td>
<td>Y/N</td>
</tr>
<tr>
<td>ADD UNNORMALIZED</td>
<td>No</td>
<td>Y/N</td>
<td>-</td>
</tr>
<tr>
<td>CONVERT BFP TO HFP¹</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>CONVERT FROM FIXED</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DIVIDE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HALVE</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>LOAD¹</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LOAD AND TEST</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>LOAD COMPLEMENT</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>LOAD FP INTEGER</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LOAD LENGENEED</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>LOAD NEGATIVE</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>LOAD POSITIVE</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>LOAD ROUNDED</td>
<td>No</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>LOAD ZERO¹</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MULTIPLY</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SQUARE ROOT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>STORE¹</td>
<td>No</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED</td>
<td>Yes</td>
<td>Y/N</td>
<td>Y/N</td>
</tr>
<tr>
<td>SUBTRACT UNNORMALIZED</td>
<td>No</td>
<td>Y/N</td>
<td>-</td>
</tr>
</tbody>
</table>

**Explanation:**
- Not applicable.
- ¹ Floating-point-support instruction.
- Y/N When the HFP-significance mask bit (PSW bit 23) is zero, a true zero is forced. When the HFP-significance mask bit is one, the characteristic remains unchanged, and a program interruption for HFP significance occurs.

*Figure 18-1. Normalization and Zero Handling for Instructions with HFP Results*

The sign of a sum, difference, product, quotient, square root, the result of CONVERT FROM FIXED, or the result of LOAD FP INTEGER with a zero fraction is positive. The sign for a zero frac-
tion resulting from other HFP operations is established from the operand sign, the same as for nonzero fractions.

Normalization

A quantity can be represented with the greatest precision by an HFP number of a given fraction length when that number is normalized. A normalized HFP number has a nonzero leftmost hexadecimal fraction digit. If one or more leftmost fraction digits are zeros, the number is said to be unnormalized.

Unnormalized numbers are normalized by shifting the fraction left, one digit at a time, until the leftmost hexadecimal digit is nonzero and reducing the characteristic by the number of hexadecimal digits shifted. A number with a zero fraction cannot be normalized; either its characteristic remains unchanged or its characteristic is made zero when the result is forced to be a true zero.

Addition and subtraction with extended operands, as well as the MULTIPLY, DIVIDE, CONVERT FROM FIXED, HALVE, LOAD FP INTEGER, and SQUARE ROOT operations, are performed only with normalization. Addition and subtraction with short or long operands may be specified as either normalized or unnormalized. For all other operations, the result is produced without normalization.

With unnormalized operations, leftmost zeros in the result fraction are not eliminated. The result may or may not be in normalized form, depending upon the original operands.

In both normalized and unnormalized operations, the initial operands need not be in normalized form. The operands for multiply, divide, and square-root operations are normalized before the arithmetic process. For other normalized operations, normalization takes place when the intermediate arithmetic result is changed to the final result.

When the intermediate result of addition, subtraction, or rounding causes the fraction to overflow, the fraction is shifted right by one hexadecimal-digit position, and the value one is supplied to the vacated leftmost digit position. The fraction is then truncated to the final result length, while the characteristic is increased by one. This adjustment is made for both normalized and unnormalized operations.

Figure 18-1 on page 18-2 summarizes, for all instructions producing HFP results, the handling of zero results and whether normalization occurs for nonzero results.

Programming Note: Up to three leftmost bits of the fraction of a normalized number may be zeros, since the nonzero test applies to the entire leftmost hexadecimal digit.

HFP Data Format

HFP numbers have a 32-bit (short) format, a 64-bit (long) format, or a 128-bit (extended) format. Numbers in the short and long formats may be designated as operands both in storage and in the floating-point registers, whereas operands having the extended format can be designated only in the floating-point registers.

In all formats, the first bit (bit 0) is the sign bit (S). The next seven bits are the characteristic. In the short and long formats, the remaining bits constitute the fraction, which consists of six or 14 hexadecimal digits, respectively.

Short HFP Number

<table>
<thead>
<tr>
<th>S</th>
<th>Characteristic</th>
<th>6-Digit Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Long HFP Number

<table>
<thead>
<tr>
<th>S</th>
<th>Characteristic</th>
<th>14-Digit Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Extended HFP Number

High-Order Part

<table>
<thead>
<tr>
<th>High-Order</th>
<th>Leftmost 14 Digits of 28-Digit Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Characteristic</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Low-Order Part

<table>
<thead>
<tr>
<th>Low-Order</th>
<th>Rightmost 14 Digits of 28-Digit Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Characteristic</td>
</tr>
<tr>
<td>64</td>
<td>72</td>
</tr>
</tbody>
</table>
An extended HFP number has a 28-digit fraction and consists of two long HFP numbers that are called the high-order and low-order parts. The high-order part may be any long HFP number. The fraction of the high-order part contains the leftmost 14 hexadecimal digits of the 28-digit fraction. The characteristic and sign of the high-order part are the characteristic and sign of the extended HFP number. If the high-order part is normalized, the extended number is considered normalized. The fraction of the low-order part contains the rightmost 14 digits of the 28-digit fraction. The sign and characteristic of the low-order part of an extended operand are ignored.

When a result is generated in the extended format and placed in a register pair, the sign of the low-order part is made the same as that of the high-order part, and, unless the result is a true zero, the low-order characteristic is made 14 less than the high-order characteristic. When the subtraction of 14 would cause the low-order characteristic to become less than zero, the characteristic is made 128 greater than its correct value. (Thus, the subtraction is performed modulo 128.) HFP exponent underflow is indicated only when the high-order characteristic underflows.

When an extended result is made a true zero, both the high-order and low-order parts are made a true zero.

The range covered by the magnitude (M) of a normalized HFP number depends on the format.

In the short format:
\[ 16^{-65} \leq M \leq (1 - 16^{-6}) \times 16^{63} \]

In the long format:
\[ 16^{-65} \leq M \leq (1 - 16^{-14}) \times 16^{63} \]

In the extended format:
\[ 16^{-65} \leq M \leq (1 - 16^{-28}) \times 16^{63} \]

In all formats, approximately:
\[ 5.4 \times 10^{-79} \leq M \leq 7.2 \times 10^{75} \]

Although the final result of an HFP operation has six hexadecimal fraction digits in the short format, 14 fraction digits in the long format, and 28 fraction digits in the extended format, intermediate results have one additional hexadecimal digit on the right. This digit is called the guard digit. The guard digit may increase the precision of the final result because it participates in addition, subtraction, and comparison operations and in the left shift that occurs during normalization.

The entire set of HFP operations with normalized results is available for short, long, and extended operands in register-register versions; and for short and long operands in register-storage versions. Most instructions generate a result that has the same format as the source operands, except that there are multiplication operations which can generate a long product from short operands or an extended product from long operands. Other exceptions are instructions which convert operands from one floating-point format to another or between floating-point and fixed-point (binary-integer) formats.

Programming Notes:

1. In the absence of an HFP exponent overflow or HFP exponent underflow, the long HFP number constituting the low-order part of an extended result correctly expresses the value of the low-order part of the extended result when the characteristic of the high-order part is 14 or higher. This applies also when the result is a true zero. When the high-order characteristic is less than 14 but the number is not a true zero, the low-order part, when considered as a long HFP number, does not express the correct characteristic value.

2. The entire fraction of an extended result participates in normalization. The low-order part alone may or may not appear to be a normalized long HFP number, depending on whether the 15th digit of the normalized 28-digit fraction is nonzero or zero.

Instructions

The HFP instructions and their mnemonics and operation codes are listed in Figure 18-2 on page 18-6. The figure indicates, in the column labeled “Characteristics,” the instruction format, when the condition code is set, the instruction fields that designate access registers, and the exceptional conditions in operand designations, data, or results that cause a program interruption.

All HFP instructions are subject to the AFP-register-control bit, bit 45 of control register 0. The AFP-register-control bit must be one when an AFP register is specified as an operand location;
otherwise, an AFP-register data exception, DXC 1, is recognized.

Mnemonics for the HFP instructions have an R as the last letter when the instruction is in the RR, RRE, or RRF format. Certain letters are used for HFP instructions to represent operand-format length and normalization, as follows:

- **F** Thirty-two-bit fixed point
- **G** Sixty-four-bit fixed point
- **D** Long normalized
- **E** Short normalized
- **U** Short unnormalized
- **W** Long unnormalized
- **X** Extended normalized

**Note:** In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designation for the assembler language are shown with each instruction. For a register-to-register operation using COMPARE (short), for example, CER is the mnemonic and \( R_1, R_2 \) the operand designation.

**Programming Note:** The following additional HFP instructions are available when the HFP-multiply-and-add/subtract facility is installed:

- **MULTIPLY AND ADD** (MAD, MADR, MAE, MAER)
- **MULTIPLY AND SUBTRACT** (MSD, MSDR, MSE, MSER)
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD NORMALIZED (extended HFP)</td>
<td>AXR</td>
<td>RR C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AER</td>
<td>RR C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (long HFP)</td>
<td>AWR</td>
<td>RR C</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (long HFP)</td>
<td>AW</td>
<td>RX C A</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (short HFP)</td>
<td>AUR</td>
<td>RX C A</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (short HFP)</td>
<td>AUR</td>
<td>RX C A</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (short HFP)</td>
<td>AUR</td>
<td>RX C A</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>AER</td>
<td>RR C</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>AER</td>
<td>RR C A</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (extended HFP)</td>
<td>AXR</td>
<td>RR C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AER</td>
<td>RR C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AD</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A</td>
<td>Da EU EO LS</td>
</tr>
<tr>
<td>Name</td>
<td>Mne-</td>
<td>Characteristics</td>
<td>Op Code</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------</td>
<td>-----------------</td>
<td>---------</td>
</tr>
<tr>
<td>LOAD FP INTEGER (long HFP)</td>
<td>FIDR</td>
<td>Da</td>
<td>B37F</td>
</tr>
<tr>
<td>LOAD FP INTEGER (short HFP)</td>
<td>FIER</td>
<td>Da</td>
<td>B377</td>
</tr>
<tr>
<td>LOAD LENGTHENED (long to ext. HFP)</td>
<td>LXDR</td>
<td>Da</td>
<td>B325</td>
</tr>
<tr>
<td>LOAD LENGTHENED (long to ext. HFP)</td>
<td>LXD</td>
<td>Da</td>
<td>B326</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>LXER</td>
<td>Da</td>
<td>B325</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>LXE</td>
<td>Da</td>
<td>B326</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>LDER</td>
<td>Da</td>
<td>B324</td>
</tr>
<tr>
<td>LOAD NEGATIVE (extended HFP)</td>
<td>LNDR</td>
<td>Da</td>
<td>B361</td>
</tr>
<tr>
<td>LOAD NEGATIVE (long HFP)</td>
<td>LNER</td>
<td>Da</td>
<td>B360</td>
</tr>
<tr>
<td>LOAD POSITIVE (extended HFP)</td>
<td>LPXR</td>
<td>Da</td>
<td>B36/zerodot</td>
</tr>
<tr>
<td>LOAD POSITIVE (long HFP)</td>
<td>LPER</td>
<td>Da</td>
<td>B/zerodot</td>
</tr>
<tr>
<td>LOAD POSITIVE (short HFP)</td>
<td>LPER</td>
<td>Da</td>
<td>B337</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to long HFP)</td>
<td>LROR</td>
<td>Da</td>
<td>B366</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to short HFP)</td>
<td>LEXR</td>
<td>Da</td>
<td>B367</td>
</tr>
<tr>
<td>LOAD ROUNDED (long to short HFP)</td>
<td>LDXR</td>
<td>Da</td>
<td>B368</td>
</tr>
<tr>
<td>LOAD ROUNDED (long to short HFP)</td>
<td>LXE</td>
<td>Da</td>
<td>B369</td>
</tr>
<tr>
<td>LOAD ROUNDED (short to long HFP)</td>
<td>MDER</td>
<td>Da</td>
<td>B370</td>
</tr>
<tr>
<td>LOAD ROUNDED (short to long HFP)</td>
<td>MEDR</td>
<td>Da</td>
<td>B371</td>
</tr>
<tr>
<td>LOAD ROUNDED (short to long HFP)</td>
<td>MEER</td>
<td>Da</td>
<td>B372</td>
</tr>
<tr>
<td>MULTIPLY (long HFP)</td>
<td>MDR</td>
<td>Da EU EO</td>
<td>B3C</td>
</tr>
<tr>
<td>MULTIPLY (short HFP)</td>
<td>MEE</td>
<td>Da EU EO</td>
<td>B2ED37</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>MADR</td>
<td>Da EU EO</td>
<td>B33E</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short HFP)</td>
<td>MAD</td>
<td>Da EU EO</td>
<td>B32E</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short HFP)</td>
<td>MAER</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>SQUARE ROOT (extended HFP)</td>
<td>SQXR</td>
<td>Da SQ</td>
<td>B336</td>
</tr>
<tr>
<td>SQUARE ROOT (long HFP)</td>
<td>SQR</td>
<td>Da SQ</td>
<td>B244</td>
</tr>
<tr>
<td>SQUARE ROOT (long HFP)</td>
<td>SQO</td>
<td>Da SQ</td>
<td>B2ED35</td>
</tr>
<tr>
<td>SQUARE ROOT (short HFP)</td>
<td>SQER</td>
<td>Da SQ</td>
<td>B245</td>
</tr>
<tr>
<td>SQUARE ROUT (short HFP)</td>
<td>SQE</td>
<td>Da SQ</td>
<td>B2ED3Id</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED (extended HFP)</td>
<td>SXR</td>
<td>Da EU EO LS</td>
<td>37</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED (long HFP)</td>
<td>SDR</td>
<td>Da EU EO LS</td>
<td>2B</td>
</tr>
</tbody>
</table>

Figure 18-2 (Part 2 of 3). Summary of HFP Instructions
### ADD NORMALIZED

**Mnemonic1**  \( R_1, R_2 \)  \([RR]\)

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

**Mnemonic1 Op Code Operands**

- **AER** '3A' Short HFP
- **ADR** '2A' Long HFP
- **AXR** '36' Extended HFP

**Mnemonic2**  \( R_1, D_2(X_2, B_2) \)  \([RX]\)

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( D_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

**Mnemonic2 Op Code Operands**

- **AE** '7A' Short HFP
- **AD** '6A' Long HFP

The second operand is added to the first operand, and the normalized sum is placed at the first-operand location.

Addition of two HFP numbers consists in characteristic comparison, fraction alignment, and signed fraction addition. The characteristics of the two operands are compared, and the fraction accompanying the smaller characteristic is aligned with the other fraction by a right shift, with its characteristic increased by one for each hexadecimal digit of shift until the two characteristics agree.

When a fraction is shifted right during alignment, the leftmost hexadecimal digit shifted out is retained as a guard digit. The fraction that is not shifted is considered to be extended with a zero in the guard-digit position. When no alignment shift occurs, both operands are considered to be extended with zeros in the guard-digit position. The fractions with signs are then added algebraically to form a signed intermediate sum.
The intermediate-sum fraction consists of seven (short format), 15 (long format), or 29 (extended format) hexadecimal digits, including the guard digit, and a possible carry. If a carry is present, the sum is shifted right one digit position so that the carry becomes the leftmost digit of the fraction, and the characteristic is increased by one.

If the addition produces no carry, the intermediate-sum fraction is shifted left as necessary to eliminate any leading hexadecimal zero digits resulting from the addition, provided the fraction is not zero. Zeros are supplied to the vacated rightmost digits, and the characteristic is reduced by the number of hexadecimal digits of shift. The fraction thus normalized is then truncated on the right to six (short format), 14 (long format), or 28 (extended format) hexadecimal digits. In the extended format, a characteristic is generated for the low-order part, which is 14 less than the high-order characteristic.

The sign of the sum is determined by the rules of algebra, unless all digits of the intermediate-sum fraction are zero, in which case the result is made a positive true zero.

An HFP-exponent-overflow exception exists when a carry from the leftmost position of the intermediate-sum fraction would cause the characteristic of the normalized sum to exceed 127. The operation is completed by making the result characteristic 128 less than the correct value, and a program interruption for HFP exponent overflow occurs. The result is normalized, and the sign and fraction remain correct. For extended results, the characteristic of the low-order part remains correct.

An HFP-exponent-underflow exception exists when the characteristic of the normalized sum would be less than zero and the fraction is not zero. If the HFP-exponent-underflow mask bit in the PSW is one, the operation is completed by making the result characteristic 128 greater than the correct value, and a program interruption for HFP exponent underflow occurs. The result is normalized, and the sign and fraction remain correct. If the HFP-exponent-underflow mask bit in the PSW is zero, a program interruption does not occur; instead, the operation is completed by making the result a positive true zero. For extended results, HFP exponent underflow is not recognized when the low-order characteristic is less than zero but the high-order characteristic is equal to or greater than zero.

The result fraction is zero when the intermediate-sum fraction, including the guard digit, is zero. With a zero result fraction, the action depends on the setting of the HFP-significance mask bit in the PSW. If the HFP-significance mask bit in the PSW is one, no normalization occurs, the intermediate and final result characteristics are the same, and a program interruption for HFP significance occurs. If the HFP-significance mask bit in the PSW is zero, the program interruption does not occur; instead, the result is made a positive true zero.

For AXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

\[
\begin{align*}
0 & \text{ Result is zero} \\
1 & \text{ Result is less than zero} \\
2 & \text{ Result is greater than zero} \\
3 & \text{ --}
\end{align*}
\]

**Program Exceptions:**

- Access (fetch, operand 2 of AE and AD only)
- Data with DXC 1, AFP register
- HFP exponent overflow
- HFP exponent underflow
- HFP significance
- Specification (AXR only)

**Programming Notes:**

1. An example of the use of the ADD NORMALIZED instruction (AE) is given in Appendix A.
2. Interchanging the two operands in an HFP addition does not affect the value of the sum.
3. The ADD NORMALIZED instruction normalizes the sum but not the operands. Thus, if one or both operands are unnormalized, precision may be lost during fraction alignment.
### ADD UNNORMALIZED

**Mnemonic 1**  \( R_1, R_2 \)  \[RR\]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

**Mnemonic 1**  

**Op Code**  

- **AUR** '3E' Short HFP
- **AWR** '2E' Long HFP

**Mnemonic 2**  \( R_1, D_2(X_2, B_2) \)  \[RX\]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( D_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

**Mnemonic 2**  

**Op Code**  

- **AU** '7E' Short HFP
- **AW** '6E' Long HFP

The second operand is added to the first operand, and the unnormalized sum is placed at the first-operand location.

The execution of ADD UNNORMALIZED is identical to that of ADD NORMALIZED, except that:

1. When no carry is present after the addition, the intermediate-sum fraction is truncated to the proper result-fraction length without a left shift to eliminate leading hexadecimal zeros and without the corresponding reduction of the characteristic.
2. HFP exponent underflow cannot occur.
3. The guard digit does not participate in the recognition of a zero result fraction. A zero result fraction is recognized when the fraction (that is, the intermediate-sum fraction, excluding the guard digit) is zero.

**Resulting Condition Code:**

- 0 Result fraction zero
- 1 Result less than zero
- 2 Result greater than zero
- 3 --

**Program Exceptions:**

- Access (fetch, operand 2 of AU and AW only)
- Data with DXC 1, AFP register
- HFP exponent overflow
- HFP significance

### COMPARE

**Mnemonic 1**  \( R_1, R_2 \)  \[RR\]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

**Mnemonic 1**  

**Op Code**  

- **CER** '39' Short HFP
- **CDR** '29' Long HFP

**Mnemonic 2**  \( R_1, R_2 \)  \[RRE\]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

**Mnemonic 2**  

**Op Code**  

- **CXR** 'B369' Extended HFP

**Mnemonic 3**  \( R_1, D_2(X_2, B_2) \)  \[RX\]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( D_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

**Mnemonic 3**  

**Op Code**  

- **CE** '79' Short HFP
- **CD** '69' Long HFP

The first operand is compared with the second operand, and the condition code is set to indicate the result.

The comparison is algebraic and follows the procedure for normalized subtraction, except that the difference is discarded after setting the condition code and both operands remain unchanged. When the difference, including the guard digit, is zero, the operands are equal. When a nonzero difference is positive or negative, the first operand is high or low, respectively.
An HFP-exponent-overflow, HFP-exponent-underflow, or HFP-significance exception cannot occur.

For CXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operands equal</td>
</tr>
<tr>
<td>1</td>
<td>First operand low</td>
</tr>
<tr>
<td>2</td>
<td>First operand high</td>
</tr>
<tr>
<td>3</td>
<td>--</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Access (fetch, operand 2 of CE and CD only)
- Data with DXC 1, AFP register
- Specification (CXR only)

**Programming Notes:**

1. Examples of the use of the COMPARE instruction (CDR) are given in Appendix A.
2. An exponent inequality alone is not sufficient to determine the inequality of two operands with the same sign, because the fractions may have different numbers of leading hexadecimal zeros.
3. Numbers with zero fractions compare equal even when they differ in sign or characteristic.

**CONVERT FROM FIXED**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>R1, R2</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEFR</td>
<td>'B3B4'</td>
<td>32-bit binary integer operand, short HFP result</td>
</tr>
<tr>
<td>CDFR</td>
<td>'B3B5'</td>
<td>32-bit binary integer operand, long HFP result</td>
</tr>
<tr>
<td>CXFR</td>
<td>'B3B6'</td>
<td>32-bit binary integer operand, extended HFP result</td>
</tr>
<tr>
<td>CEGR</td>
<td>'B3C4'</td>
<td>64-bit binary integer operand, short HFP result</td>
</tr>
<tr>
<td>CDGR</td>
<td>'B3C5'</td>
<td>64-bit binary integer operand, long HFP result</td>
</tr>
<tr>
<td>CXGR</td>
<td>'B3C6'</td>
<td>64-bit binary integer operand, extended HFP result</td>
</tr>
</tbody>
</table>

The fixed-point second operand is converted to the HFP format, and the normalized result is placed at the first-operand location.

A nonzero result is normalized. A zero result is made a positive true zero.

The second operand is a signed binary integer that is located in the general register designated by R2. A 32-bit operand is in bit positions 32-63 of the register.

The result is normalized and rounded toward zero (truncated) before it is placed at the first-operand location.

For CXFR and CXGR, the R1 field must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Data with DXC 1, AFP register
- Specification (CXFR and CXGR)

**CONVERT TO FIXED**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>R1, M3, R2</th>
<th>[RRF]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>M3</td>
<td>R1</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFER</td>
<td>'B3B8'</td>
<td>Short HFP operand, 32-bit binary-integer result</td>
</tr>
<tr>
<td>CFDR</td>
<td>'B3B9'</td>
<td>Long HFP operand, 32-bit binary-integer result</td>
</tr>
<tr>
<td>CFXR</td>
<td>'B3BA'</td>
<td>Extended HFP operand, 32-bit binary-integer result</td>
</tr>
<tr>
<td>CGER</td>
<td>'B3C8'</td>
<td>Short HFP operand, 64-bit binary-integer result</td>
</tr>
<tr>
<td>CGDR</td>
<td>'B3C9'</td>
<td>Long HFP operand, 64-bit binary-integer result</td>
</tr>
<tr>
<td>CGXR</td>
<td>'B3CA'</td>
<td>Extended HFP operand, 64-bit binary-integer result</td>
</tr>
</tbody>
</table>

The HFP second operand is rounded to an integer value and then converted to the fixed-point format. The result is placed at the first-operand location.

The result is a signed binary integer that is placed in the general register designated by R1. A 32-bit result replaces bits 32-63 of the register, and bits 0-31 of the register remain unchanged.
The second operand is rounded to an integer value by rounding as specified by the modifier in the M₃ field:

**M₃ Rounding Method**

- 0: Round toward 0
- 1: Biased round to nearest
- 4: Round to nearest
- 5: Round toward 0
- 6: Round toward +∞
- 7: Round toward −∞

A modifier other than 0, 1, or 4-7 is invalid.

The sign of the result is the sign of the second operand, except that a zero result has a plus sign.

If the rounded result would have a value exceeding the range that can be represented in the result format, the largest (in magnitude) representable number of the same sign as the source is placed at the target location, and condition code 3 is set.

HFP exponent underflow is not recognized because small values are rounded to one (with the appropriate sign) or to zero, depending on the rounding mode.

The M₃ field must designate a valid modifier; otherwise, a specification exception is recognized. For CFXR and CGXR, the R₂ field must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

- 0: Source was zero
- 1: Source was less than zero
- 2: Source was greater than zero
- 3: Special case

**Program Exceptions:**

- Data with DXC 1, AFP register
- Specification

---

### DIVIDE

#### Mnemonic 1

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

#### Mnemonic 1 Op Code Operands

- DER '3D': Short HFP
- DDR '2D': Long HFP

#### Mnemonic 2

<table>
<thead>
<tr>
<th>'B22D'</th>
<th>///////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

#### Mnemonic 2 Op Code Operands

- DXR 'B22D': Extended HFP

#### Mnemonic 3

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

#### Mnemonic 3 Op Code Operands

- DE '7D': Short HFP
- DD '6D': Long HFP

The first operand (the dividend) is divided by the second operand (the divisor), and the normalized quotient is placed at the first-operand location. No remainder is preserved.

HFP division consists in characteristic subtraction and fraction division. The operands are first normalized to eliminate leading hexadecimal zeros. The difference between the dividend and divisor characteristics of the normalized operands, plus 64, is used as the characteristic of an intermediate quotient.

All dividend and divisor fraction digits participate in forming the fraction of the intermediate quotient. The intermediate-quotient fraction can have no leading hexadecimal zeros, but a right shift of one digit position may be necessary, with this causing an increase of the characteristic by one. The fraction is then truncated to the proper result-fraction length.

An HFP-exponent-overflow exception exists when the characteristic of the final quotient would exceed 127 and the fraction is not zero. The operation is completed by making the result characteristic 128 less than the correct value, and a
program interruption for HFP exponent overflow occurs. The result is normalized, and the sign and fraction remain correct. If, for extended results, the low-order characteristic would also exceed 127, it too is decreased by 128.

An HFP-exponent-underflow exception exists when the characteristic of the final quotient would be less than zero and the fraction is not zero. If the HFP-exponent-underflow mask bit in the PSW is one, the operation is completed by making the result characteristic 128 greater than the correct value, and a program interruption for HFP exponent underflow occurs. The result is normalized, and the sign and fraction remain correct. If the HFP-exponent-underflow mask bit in the PSW is zero, a program interruption does not occur; instead, the operation is completed by making the result a positive true zero. For extended results, HFP exponent underflow is not recognized when the low-order characteristic is less than zero but the high-order characteristic is equal to or greater than zero.

HFP exponent underflow does not occur when the characteristic of an operand becomes less than zero during normalization of the operands or when the intermediate-quotient characteristic is less than zero, as long as the final quotient can be represented with the correct characteristic.

When the divisor fraction is zero, an HFP-divide exception is recognized. This includes the case of division of zero by zero.

When the dividend fraction is zero but the divisor fraction is nonzero, the quotient is made a positive true zero. No HFP exponent overflow or HFP exponent underflow occurs.

The sign of the quotient is the exclusive or of the operand signs, except that the sign is always plus when the quotient is made a positive true zero.

For DXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2 of DE and DD only)
- Data with DXC 1, AFP register
- HFP divide
- HFP exponent overflow
- HFP exponent underflow
- Specification (DXR only)

Programming Note: Examples of the use of the DIVIDE instruction (DER) are given in Appendix A.

HALVE

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>R1, R2</th>
<th>[RR]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
HER '34' Short HFP
HDR '24' Long HFP

The second operand is divided by 2, and the normalized quotient is placed at the first-operand location.

The fraction of the second operand is shifted right one bit position, placing the contents of the rightmost bit position of the guard digit, and a zero is supplied to the leftmost bit position of the fraction. The intermediate result, including the guard digit, is then normalized, and the final result is truncated to the proper length.

An HFP-exponent-underflow exception exists when the characteristic of the final result would be less than zero and the fraction is not zero. If the HFP-exponent-underflow mask bit in the PSW is one, the operation is completed by making the result characteristic 128 greater than the correct value, and a program interruption for HFP exponent underflow occurs. The result is normalized, and the sign and fraction remain correct. If the HFP-exponent-underflow mask bit in the PSW is zero, a program interruption does not occur; instead, the operation is completed by making the result a positive true zero.

When the fraction of the second operand is zero, the result is made a positive true zero, and no HFP exponent underflow occurs.
The sign of the result is the same as that of the second operand, except that the sign is always plus when the quotient is made a positive true zero.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Data with DXC 1, AFP register
- HFP exponent underflow

**Programming Notes:**
1. An example of the use of the HALVE instruction (HDR) is given in Appendix A.
2. With short and long operands, the halve operation is identical to a divide operation with the number 2 as divisor. Similarly, the result of HDR is identical to that of MD or MDR with one-half as a multiplier, and the result of HER is identical to that of MEE or MEER with one-half as a multiplier.

**LOAD AND TEST**

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>R1,R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>R1</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTER</td>
<td>'32'</td>
<td>Short HFP</td>
</tr>
<tr>
<td>LTDR</td>
<td>'22'</td>
<td>Long HFP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic2</th>
<th>R1,R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>R1</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic2</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTXR</td>
<td>'B362'</td>
<td>Extended HFP</td>
</tr>
</tbody>
</table>

The second operand is placed at the first-operand location, and its sign and magnitude are tested to determine the setting of the condition code. The condition code is set the same as for a comparison of the second operand with zero.

For short and long operands, the second operand is placed unchanged in the first-operand location.

For extended operands, the high-order sign and the entire fraction of the source are placed unchanged in the result, and the low-order sign is set equal to the high-order sign. If the extended-operand fraction is nonzero, the high-order characteristic is placed unchanged in the result high-order characteristic, and the low-order characteristic is set to 14 less than the high-order characteristic, modulo 128. If the extended-operand fraction is zero, the result is made a true zero with the same sign as the source (the high-order and low-order sign bits of the result are the same as the high-order sign bit of the source).

For LTXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**
- 0 Result is zero
- 1 Result is less than zero
- 2 Result is greater than zero
- 3 --

**Program Exceptions:**
- Data with DXC 1, AFP register
- Specification (LTXR only)

**Programming Note:** When, for LTER and LTDR, the same register is designated as the first-operand and second-operand location, the operation is equivalent to a test without data movement.

**LOAD COMPLEMENT**

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>R1,R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>R1</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCER</td>
<td>'33'</td>
<td>Short HFP</td>
</tr>
<tr>
<td>LCDR</td>
<td>'23'</td>
<td>Long HFP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic2</th>
<th>R1,R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>R1</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic2</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCXR</td>
<td>'B363'</td>
<td>Extended HFP</td>
</tr>
</tbody>
</table>

The second operand is placed at the first-operand location with the sign bit inverted.
The sign bit is inverted even if the operand is zero. For all operand lengths, the source fraction is placed unchanged in the result.

For short and long operands, the source characteristic is placed unchanged in the result.

For extended operands, the low-order sign is set equal to the high-order sign. If the extended-operand fraction is nonzero, the high-order characteristic is set to 14 less than the high-order characteristic, modulo 128. If the extended-operand fraction is zero, the result is made a true zero with the sign inverted from the source (the high-order and low-order sign bits of the result are inverted from the high-order sign bit of the source).

For LCXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Result is zero</td>
</tr>
<tr>
<td>1</td>
<td>Result is less than zero</td>
</tr>
<tr>
<td>2</td>
<td>Result is greater than zero</td>
</tr>
<tr>
<td>3</td>
<td>--</td>
</tr>
</tbody>
</table>

**Program Exceptions:**

- Data with DXC 1, AFP register
- Specification (LCXR only)

**LOAD FP INTEGER**

Mnemonic R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

**LOAD LENGTHENED**

Mnemonic1 R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Mnemonic2 R1,Dz(X2,B2) [RXE]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40 47</td>
</tr>
</tbody>
</table>

For FIXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Data with DXC 1, AFP register
- Specification (FIXR only)

**Programming Notes:**

1. LOAD FP INTEGER truncates (rounds toward zero) an HFP number to an integer value. These integers, which remain in the HFP format, should not be confused with binary integers, which use a fixed-point format.
2. If the HFP operand is numeric with a large enough exponent so that it is already an integer, the result value remains the same, except that an unnormalized operand is normalized, and an operand with a zero fraction is changed to a positive true zero.

The second operand is truncated (rounded toward zero) to an integer value in the same floating-point format, and the normalized result is placed at the first-operand location.

A nonzero result is normalized. A zero result is made a positive true zero.
For all operand lengths, the source fraction is extended with zeros and placed in the result. The sign bit of the result is set the same as the sign of the source even when the result is made a true zero.

For long results, the source characteristic is placed unchanged in the result.

For extended results, the low-order sign is set equal to the high-order sign. If the fraction is nonzero, the source characteristic is placed unchanged in the result high-order characteristic, and the low-order characteristic is set to 14 less than the high-order characteristic, modulo 128. If the fraction is zero, the result is made a true zero with the same sign as the source (the high-order and low-order sign bits of the result are the same as the sign bit of the source).

For LXD, LXDR, LXE, and LXER, the Ri field must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of LDE, LXE, and LXD only)
- Data with DXC 1, AFP register
- Specification (LXE, LXER, LXD, LXDR)

### LOAD NEGATIVE

**Mnemonic1** R1,R2 [RR]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

**Mnemonic1** Op Code Operands
- LNER '31' Short HFP
- LNDR '21' Long HFP

**Mnemonic2** R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

**Mnemonic2** Op Code Operands
- LNXR 'B361' Extended HFP

The second operand is placed at the first-operand location with the sign bit made one.

The sign bit is made one even if the operand is zero. For all operand lengths, the source fraction is placed unchanged in the result.

For short and long operands, the source characteristic is placed unchanged in the result.

For extended operands, the low-order sign is set equal to the high-order sign. If the extended-operand fraction is nonzero, the high-order characteristic is placed unchanged in the result high-order characteristic, and the low-order characteristic is set to 14 less than the high-order characteristic, modulo 128. If the extended-operand fraction is zero, the result is made a negative true zero (the high-order and low-order sign bits of the result are set to one).

For LNXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**
- 0 Result is zero
- 1 Result is less than zero
- 2 --
- 3 --

**Program Exceptions:**
- Data with DXC 1, AFP register
- Specification (LNXR only)

### LOAD POSITIVE

**Mnemonic1** R1,R2 [RR]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

**Mnemonic1** Op Code Operands
- LPER '3/zerodot' Short HFP
- LPDR '2/zerodot' Long HFP

**Mnemonic2** R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

**Mnemonic2** Op Code Operands
- LPXR 'B36/zerodot' Extended HFP

The second operand is placed at the first-operand location with the sign bit made zero.
For all operand lengths, the sign bit is made zero, and the source fraction is placed unchanged in the result.

For short and long operands, the source characteristic is placed unchanged in the result.

For extended operands, the low-order sign is set equal to the high-order sign. If the extended-operand fraction is nonzero, the high-order characteristic is set to 14 less than the high-order characteristic, modulo 128. If the extended-operand fraction is zero, the result is made a positive true zero (the high-order and low-order sign bits of the result are set to zero).

For LPXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0  Result is zero
1  --
2  Result is greater than zero
3  --

**Program Exceptions:**

- Data with DXC 1, AFP register
- Specification (LPXR only)

**LOAD ROUNDED**

Mnemonic1  R1,R2  [RR]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Mnemonic1  Op Code  Operands

<table>
<thead>
<tr>
<th>LEDR</th>
<th>'35'</th>
<th>Long HFP operand 2, short HFP operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDXR</td>
<td>'25'</td>
<td>Extended HFP operand 2, long HFP operand 1</td>
</tr>
</tbody>
</table>

The above mnemonics are alternatives to the following older mnemonics that are less descriptive of operand lengths:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRER</td>
<td>'35'</td>
<td>Long HFP operand 2, short HFP operand 1</td>
</tr>
<tr>
<td>LRDR</td>
<td>'25'</td>
<td>Extended HFP operand 2, long HFP operand 1</td>
</tr>
</tbody>
</table>

The second operand is rounded to a shorter format, and the result is placed at the first-operand location.

Rounding consists in adding a one to the leftmost bit position of the second operand that is to be dropped and propagating any carry through the fraction. The sign of the second operand is ignored, and addition is performed as if the fraction were positive.

If rounding causes a carry out of the leftmost hexadecimal digit position of the fraction, the fraction is shifted right one digit position so that the carry becomes the leftmost digit of the fraction, and the characteristic is increased by one.

The intermediate fraction is then truncated to the proper result-fraction length. For LEDR and LDXR, the result replaces the leftmost 32 bits of the target register, and the rightmost 32 bit positions of the target register remain unchanged. For LDXR, the 64-bit result is placed in a floating-point register, not a floating-point register pair.

The sign of the result is the same as the sign of the second operand. There is no normalization to eliminate leading zeros.

An HFP-exponent-overflow exception exists when shifting the fraction right would cause the characteristic to exceed 127. The operation is completed by making the result characteristic 128 less than the correct value, and a program interruption for HFP exponent overflow occurs. The result is normalized, and the sign and fraction remain correct.

HFP-exponent-underflow and HFP-significance exceptions cannot occur.

For LDXR (or LRDR) and LEXR, the R2 field must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.
Program Exceptions:
- Data with DXC 1, AFP register
- HFP exponent overflow
- Specification (LDXR, LEXR, LRDR)

Programming Note: The sign of the rounded result is the same as the sign of the operand, even when the result is zero.

MULTIPLY

**Mnemonic1** R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>//////////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

**Mnemonic1 Op Code** Operands
MEER 'B337' Short HFP

**Mnemonic2** R₁,R₂  [RR]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

**Mnemonic2 Op Code** Operands
MDR '2C' Long HFP
MXR '26' Extended HFP
MDER '3C' Short HFP multiplier and multiplicand, long HFP product
MXDR '27' Long HFP multiplier and multiplicand, extended HFP product

The above mnemonic MDER is an alternative to the following older mnemonic that is less descriptive of operand lengths:
ME '7C' Short HFP multiplier and multiplicand, long HFP product

The normalized product of the second operand (the multiplier) and the first operand (the multiplicand) is placed at the first-operand location.

Multiplication of two HFP numbers consists in exponent addition and fraction multiplication. The operands are first normalized to eliminate leading hexadecimal zeros. The sum of the characteristics of the normalized operands, less 64, is used as the characteristic of the intermediate product.

The fraction of the intermediate product is the exact product of the normalized operand fractions. If the intermediate-product fraction has one leading hexadecimal zero digit, the fraction is shifted left one digit position, bringing the contents of the guard-digit position into the rightmost position of the result fraction, and the intermediate-product characteristic is reduced by one. The fraction is then truncated to the proper result-fraction length.

For MDE and MDER, the multiplier and multiplicand fractions have six hexadecimal digits; the product fraction has the full 14 digits of the long format, with the two rightmost fraction digits always zeros. For MEE and MEER, the multiplier and multiplicand fractions have six digits, and the final product fraction is truncated to six digits; the result, as for all short-format results, replaces the leftmost 32 bits of the target register, and the rightmost 32 bit positions of the target register remain unchanged.

For MD and MDR, the multiplier and multiplicand fractions have 14 digits, and the final product frac-
tion is truncated to 14 digits. For MXD and MXDR, the multiplier and multiplicand fractions have 14 digits, with the multiplicand occupying the high-order part of the first operand; the final product fraction contains 28 digits and is an exact product of the operand fractions. For MXR, the multiplier and multiplicand fractions have 28 digits, and the final product fraction is truncated to 28 digits.

An HFP-exponent-overflow exception exists when the characteristic of the final product would exceed 127 and the fraction is not zero. The operation is completed by making the result characteristic 128 less than the correct value, and a program interruption for HFP exponent overflow occurs. The result is normalized, and the sign and fraction remain correct. If, for extended results, the low-order characteristic would also exceed 127, it too is decreased by 128.

HFP exponent overflow is not recognized when the intermediate-product characteristic is initially 128 but is brought back within range by normalization.

An HFP-exponent-underflow exception exists when the characteristic of the final product would be less than zero and the fraction is not zero. If the HFP-exponent-underflow mask bit in the PSW is one, the operation is completed by making the result characteristic 128 greater than the correct value, and a program interruption for HFP exponent underflow occurs. The result is normalized, and the sign and fraction remain correct. If the HFP-exponent-underflow mask bit in the PSW is zero, a program interruption does not occur; instead, the operation is completed by making the result a positive true zero. For extended results, HFP exponent underflow is not recognized when the low-order characteristic is less than zero but the high-order characteristic is equal to or greater than zero.

HFP exponent underflow does not occur when the characteristic of an operand becomes less than zero during normalization of the operands, as long as the final product can be represented with the correct characteristic.

If either or both operand fractions are zero, the result is made a positive true zero, and no HFP exponent overflow or HFP exponent underflow occurs.

The sign of the product is the exclusive or of the operand signs, except that the sign is always plus when the result is made a true zero.

The R1 field for MXD, MXDR, and MXR, and the R2 field for MXR must designate valid floating-point-register pairs. Otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**
- Access (fetch, operand 2 of MDE, MEE, MD, and MXD only)
- Data with DXC 1, AFP register
- HFP exponent overflow
- HFP exponent underflow
- Specification (MXD, MXDR, MXR)

**Programming Notes:**
1. An example of the use of the MULTIPLY instruction (MDR) is given in Appendix A.
2. Interchanging the two operands in an HFP multiplication does not affect the value of the product.

**MULTIPLY AND ADD**

### Mnemonic 1

**R1,R3,R2**  
**[RRF]**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>///</th>
<th>R3</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

### Mnemonic 1 Operands
- **MAER** 'B32E' Short HFP
- **MADR** 'B33E' Long HFP

### Mnemonic 2

**R1,R3,D2(X2,B2)**  
**[RXF]**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
<td>40</td>
</tr>
</tbody>
</table>

### Mnemonic 2 Operands
- **MAE** 'ED2E' Short HFP
- **MAD** 'ED3E' Long HFP
MULTIPLY AND SUBTRACT

Mnemonic1 \( R_1, R_3, R_2 \)  [RRF]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 )</th>
<th>( '/', '/' )</th>
<th>( R_3 )</th>
<th>( R_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

Mnemonic1 Op Code Operands

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_3 )</th>
<th>( X_2 )</th>
<th>( B_2 )</th>
<th>( D_2 )</th>
<th>( R_1 )</th>
<th>( '/', '/' )</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>36</td>
<td>40</td>
</tr>
</tbody>
</table>

Mnemonic2 Op Code Operands

<table>
<thead>
<tr>
<th>Op Code</th>
<th>( R_1 ), ( R_3 ), ( D_2 ) (( X_2, B_2 ))</th>
<th>( X_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE</td>
<td>('ED2F') Short HFP</td>
<td></td>
</tr>
<tr>
<td>MSD</td>
<td>('ED3F') Long HFP</td>
<td></td>
</tr>
</tbody>
</table>

The third operand is multiplied by the second operand, and then the first operand is added to or subtracted from the product. The sum or difference is placed at the first-operand location. The MULTIPLY AND ADD and MULTIPLY AND SUBTRACT operations may be summarized as:

\[ op_1 = op_3 \cdot op_2 \pm op_1 \]

The third and second HFP operands are multiplied, forming an intermediate product, and the first operand is then added (or subtracted) algebraically to (or from) the intermediate product, forming an intermediate result. The exponent and fraction of the intermediate product and intermediate result are maintained exactly. The intermediate result, if nonzero, is normalized and truncated to the operand format and then placed at the first-operand location.

The sign of the result is determined by the rules of algebra, unless the intermediate-result fraction is zero, in which case the result is made a positive true zero.

An HFP-exponent-overflow exception exists when the characteristic of the normalized result would exceed 127 and the fraction is not zero. The operation is completed by making the result characteristic 128 less than the correct value, and a program interruption for HFP exponent overflow occurs. The result is normalized, and the sign and fraction remain correct.

HFP exponent overflow is not recognized on intermediate values, provided the normalized result can be represented with the correct characteristic.

An HFP-exponent-underflow exception exists when the characteristic of the normalized result would be less than zero and the fraction is not zero. If the HFP-exponent-underflow mask bit in the PSW is one, the operation is completed by making the result characteristic 128 greater than the correct value, and a program interruption for HFP exponent underflow occurs. The result is normalized, and the sign and fraction remain correct. If the HFP-exponent-underflow mask bit in the PSW is zero, a program interruption does not occur; instead, the operation is completed by making the result a positive true zero.

HFP exponent underflow is not recognized on input operands and intermediate values, provided the normalized result can be represented with the correct characteristic.

**Condition Code:** The code remains unchanged.

**Program Exceptions:**

- Access (fetch, operand 2 of MAE, MAD, MSE, MSD)
- Data with DXC 1, AFP register
- HFP exponent overflow
- HFP exponent underflow
- Operation (if the HFP multiply-add/subtract facility is not installed)

**Programming Note:** MULTIPLY AND ADD (SUBTRACT) differs from MULTIPLY followed by ADD (SUBTRACT) NORMALIZED in the following ways:

1. The product is maintained to full precision, and overflow and underflow are not recognized on the product.

2. The HFP-significance exception is not recognized for MULTIPLY AND ADD (SUBTRACT).

3. ADD (SUBTRACT) NORMALIZED maintains only a single guard digit and does not prenormalize input operands; thus, in some cases, an unnormalized input operand may cause loss of precision in the result. MULTIPLY AND ADD (SUBTRACT) maintains the entire intermediate sum (difference), which is normalized before the truncation operation is performed; thus, unnormalized operands do not cause any additional loss of precision.
4. On most models, the execution time of MULTIPLY AND ADD (SUBTRACT) is less than the combined execution time of MULTIPLY followed by ADD (SUBTRACT) NORMALIZED. The performance of MULTIPLY AND ADD (SUBTRACT) may be severely degraded in the case of unnormalized input operands.

SQUARE ROOT

Mnemonic1 R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>16</th>
<th>24</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
</table>

Mnemonic1 Op Code Operands
SQER 'B245' Short HFP
SQDR 'B244' Long HFP
SQXR 'B336' Extended HFP

Mnemonic2 R1,D2(X2,B2) [RXE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
<th>16</th>
<th>24</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
</table>

Mnemonic2 Op Code Operands
SQE 'ED34' Short HFP
SQD 'ED35' Long HFP

The normalized and rounded square root of the second operand is placed at the first-operand location.

When the fraction of the second operand is zero, the sign and characteristic of the second operand are ignored, and the operation is completed by placing a positive true zero at the first-operand location.

If the second operand is less than zero, an HFP-square-root exception is recognized.

If the second operand is normalized and greater than zero, the operation then proceeds as for normalized operands.

For SQXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

Program Exceptions:
- Access (fetch, operand 2 of SQE and SQD only)
- Data with DXC 1, AFP register
- HFP square root
- Specification (SQXR only)

Programming Notes:

1. The use of the SQUARE ROOT instruction with short operands (SQER) is illustrated by the examples in the following table:

<table>
<thead>
<tr>
<th>Operand (hex)</th>
<th>Decimal Value</th>
<th>Result (hex)</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>42 190000</td>
<td>25.0</td>
<td>41 500000</td>
<td>5.0</td>
</tr>
<tr>
<td>40 400000</td>
<td>0.250</td>
<td>40 800000</td>
<td>0.50</td>
</tr>
<tr>
<td>40 800000</td>
<td>0.50</td>
<td>40 B504F3</td>
<td>0.7071...</td>
</tr>
<tr>
<td>41 800000</td>
<td>8.0</td>
<td>41 2D4130</td>
<td>2.8284...</td>
</tr>
</tbody>
</table>

2. The result fraction is correctly normalized without any further left or right shifts of the intermediate-result fraction and without any further exponent adjustment. Rounding cannot cause a carry out of the leftmost digit.

3. Although a characteristic greater than 127 or less than zero may temporarily be generated...
during the operation, the result characteristic is always within the representable range, and no HFP exponent overflow or underflow occurs.

Specifically, the smallest nonzero operand in the long format consists of a one bit, preceded on the left by 63 zeros. This operand is an unnormalized number with a value of $16^{-78}$, and its square root is $16^{-39}$. The normalized representation of this result has a characteristic of 26 (decimal). Similarly, the square root of the largest representable operand has a characteristic of 96 (decimal). The instruction, therefore, cannot produce a nonzero result with a characteristic outside the range of 26 to 96.

**SUBTRACT NORMALIZED**

Mnemonic1

```
Mnemonic1   R₁,R₂    [RR]
```

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Mnemonic1

```
Op Code  Operands
SER '3B' Short HFP
SDR '2B' Long HFP
SXR '37' Extended HFP
```

Mnemonic2

```
Mnemonic2   R₁,D₂(X₁,B₂)    [RX]
```

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>X₁</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Mnemonic2

```
Op Code  Operands
SE '7B' Short HFP
SD '6B' Long HFP
```

The second operand is subtracted from the first operand, and the normalized difference is placed at the first-operand location.

The execution of SUBTRACT NORMALIZED is identical to that of ADD NORMALIZED, except that the second operand participates in the operation with its sign bit inverted.

For SXR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0  Result is zero
1  Result is less than zero
2  Result is greater than zero
3  --

**Program Exceptions:**

- Access (fetch, operand 2 of SE and SD only)
- Data with DXC 1, AFP register
- HFP exponent overflow
- HFP exponent underflow
- HFP significance
- Specification (SXR only)

**SUBTRACT UNNORMALIZED**

Mnemonic1

```
Mnemonic1   R₁,R₂    [RR]
```

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Mnemonic1

```
Op Code  Operands
SUR '3F' Short HFP
SWR '2F' Long HFP
```

Mnemonic2

```
Mnemonic2   R₁,D₂(X₁,B₂)    [RX]
```

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>X₁</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Mnemonic2

```
Op Code  Operands
SE '7F' Short HFP
SW '6F' Long HFP
```

The second operand is subtracted from the first operand, and the unnormalized difference is placed at the first-operand location.

The execution of SUBTRACT UNNORMALIZED is identical to that of ADD UNNORMALIZED, except that the second operand participates in the operation with its sign bit inverted.

**Resulting Condition Code:**

0  Result fraction zero
1  Result less than zero
2  Result greater than zero
3  --
Program Exceptions:

- Access (fetch, operand 2 of SU and SW only)
- Data with DXC 1, AFP register

- HFP exponent overflow
- HFP significance
Chapter 19. Binary-Floating-Point Instructions

Binary-Floating-Point Facility ........................................ 19-1
Floating-Point-Control (FPC) Register ......................... 19-2
IEEE Masks and Flags ........................................... 19-3
FPC DXC Byte .................................................. 19-3
Operations on the FPC Register .................................. 19-3
BFP Arithmetic ................................................. 19-4
BFP Data Formats ............................................... 19-4
BFP Short Format .............................................. 19-4
BFP Long Format ............................................... 19-4
BFP Extended Format .......................................... 19-4
Biased Exponent ................................................ 19-4
Significand ...................................................... 19-4
Values of Nonzero Numbers ...................................... 19-4
Classes of BFP Data ............................................. 19-5
Zeros ............................................................ 19-6
Denormalized Numbers ......................................... 19-6
Normalized Numbers .......................................... 19-6
Infinities ........................................................ 19-6
Signaling and Quiet NaNs ........................................ 19-6
BFP-Format Conversion ......................................... 19-7
BFP Rounding .................................................... 19-7
Rounding Mode .................................................. 19-7
Normalization and Denormalization ............................ 19-8
BFP Comparison .................................................. 19-8
Condition Codes for BFP Instructions ......................... 19-9
Remainder ........................................................ 19-9
IEEE Exception Conditions ...................................... 19-10
IEEE Invalid Operation ........................................ 19-10
IEEE Division-By-Zero ......................................... 19-11
IEEE Overflow .................................................. 19-11
IEEE Underflow ................................................ 19-12
IEEE Inexact ..................................................... 19-12
Result Figures ................................................... 19-13
Data-Exception Codes (DXC) and
Abbreviations .................................................... 19-14
Instructions ....................................................... 19-14
ADD ............................................................ 19-18
COMPARE ......................................................... 19-23
COMPARE AND SIGNAL ...................................... 19-24
CONVERT FROM FIXED ....................................... 19-26
CONVERT TO FIXED .......................................... 19-26
DIVIDE .......................................................... 19-29
DIVIDE TO INTEGER .......................................... 19-29
EXTRACT FPC .................................................... 19-33
LOAD AND TEST ................................................ 19-34
LOAD COMPLEMENT .......................................... 19-34
LOAD FP INTEGER ............................................. 19-35
LOAD FPC ........................................................ 19-36
LOAD LENGTHENED ............................................ 19-37
LOAD NEGATIVE ............................................... 19-37
LOAD POSITIVE ................................................ 19-38
LOAD ROUNDED ............................................... 19-38
MULTIPLY ........................................................ 19-39
MULTIPLY AND ADD .......................................... 19-41
MULTIPLY AND SUBTRACT .................................... 19-41
SET FPC .......................................................... 19-43
SET ROUNDING MODE ....................................... 19-43
SQUARE ROOT .................................................. 19-44
STORE FPC ....................................................... 19-44
SUBTRACT ........................................................ 19-44
TEST DATA CLASS ............................................. 19-45

Binary-Floating-Point Facility

The binary-floating-point (BFP) facility provides instructions to operate on binary (radix-2) floating-point data.

BFP provides a number of important advantages over hexadecimal floating point (HFP):

- Greater precision and exponent range (except for numbers in the short format where HFP has the greater range).
- Automatic rounding to the nearest value for all arithmetic operations. There are directed-rounding options that may be used instead.
- Special entities of “infinity” and “Not-a-Number” (NaN), which are accepted and handled by arithmetic operations in a reasonable fashion. They provide better defaults for exponent overflow and invalid operations (such as division of zero by zero). This allows most programs to continue running without hiding such errors and without using specialized exception handlers.
- Exponent underflow gives “denormalized” numbers as the default, which provides more consistent results than the abrupt result of zero produced by the HFP instructions.
- The greater exponent range makes exponent overflow and underflow in correctly written
programs very unlikely, so that programmers may often be able to ignore these conditions.

- Both mask and flag bits are provided for all arithmetic exception conditions. The mask bits enable or disable interruptions. When interruptions are disabled, the flag bits keep track of exception conditions during execution so that warning messages may be issued.

- Programs can be migrated from and to workstations and other systems using different architectures and still give consistent results, provided that floating-point operations on the other systems also conform to the IEEE standard. This does not mean, however, that bit-wise compatible results can be guaranteed, because the standard allows implementation flexibility, especially in the presence of exceptions.

Programming Note: The bit representation of the BFP data formats in storage is defined to be left-to-right in a manner that is uniform for all numeric operands in the z/Architecture architecture. Although the format diagrams in the IEEE floating-point standard appear to use the same left-to-right bit sequence, the standard only defines the meaning of the bits without specifying how they appear in storage; the storage arrangement is left to the implementation. Several implementations in fact use other sequences; this may affect programs which are dependent on the bit representation of floating-point data in storage.

Floating-Point-Control (FPC) Register

The floating-point-control (FPC) register is a 32-bit register that contains mask bits, flag bits, a data-exception code, and rounding-mode bits. An overview of the FPC register is shown in Figure 19-1. Details are shown in Figure 19-2 on page 19-3 and in Figure 19-3 on page 19-3. (In Figure 19-2, the abbreviations “IM” and “SF” are based on the terms “interruption mask” and “status flag,” respectively.)

The bits of the FPC register are often referred to as, for example, FPC 1.0, meaning bit 0 of byte 1 of the register.

![Figure 19-1. FPC Register Overview](image-url)
The FPC register contains five IEEE mask bits and five IEEE flag bits that each correspond to one of the five arithmetic exception conditions that may occur when a BFP instruction is executed. The mask bits, when one, cause an interruption to occur if an exception condition is recognized. If the mask bit for an exception condition is zero, the recognition of the condition causes the corresponding flag bit to be set to one. Thus, a flag bit indicates whether the corresponding exception condition has been recognized at least once since the program last set the flag bit to zero. The mask bits are ignored, and the flag bits remain unchanged, when arithmetic exceptions are recognized for floating-point-support (FPS) and HFP instructions.

The IEEE flag bits in the FPC register are set to zero only by explicit program action, clear reset, or power-on reset.

**FPC DXC Byte**

Byte 2 of the FPC register contains the data-exception code (DXC), which is an eight-bit code indicating the specific cause of a data exception. When the AFP-register-control bit, bit 45 of control register 0, is one and a program interruption causes the DXC to be placed at real location 147, the DXC is also placed in the DXC field of the FPC register. The DXC field in the FPC register remains unchanged when the AFP-register-control bit is zero or when any other program exception is reported. The DXC is described in "Data-Exception Code (DXC)" on page 6-14.

The DXC is a code, meaning it should be treated as an integer rather than as individual bits. However, when bits 6 and 7 are zero, bits 0-5 are bit significant; bits 0-4 (:i,z,o,u,x) are trap flags and correspond to the same bits in bytes 0 and 1 of the FPC register (IEEE masks and IEEE flags), and bit 5 (y) is used in conjunction with bit 4, inexact (x), to indicate that the result has been incremented in magnitude. The trap flag for an exception, instead of the IEEE flag, is set to one when an interruption for the exception is enabled by the corresponding IEEE mask bit.

**Operations on the FPC Register**

The following unprivileged BFP instructions allow problem-state programs to operate on the FPC register:

- **EXTRACT FPC**
- **LOAD FPC**
- **SET FPC**
- **SET ROUNдинG MODE**
- **STORE FPC**

These instructions are subject to the AFP-register-control bit, bit 45 of control register 0. An attempt to execute any of the above instructions when the AFP-register-control bit is zero results in a BFP-instruction data exception, DXC 2.
BFP Arithmetic

BFP Data Formats

Binary-floating-point numbers and NaNs may be represented in any of three formats: short, long, or extended.

BFP Short Format

<table>
<thead>
<tr>
<th>Exponent + 127</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8 bits)</td>
<td>(23 bits)</td>
</tr>
<tr>
<td>0 1</td>
<td>9 31</td>
</tr>
</tbody>
</table>

Figure 19-4. BFP Short Format (4 bytes)

When a number or NaN in the BFP short format is loaded into a floating-point register, it occupies the left half of the register, and the right half remains unchanged.

BFP Long Format

<table>
<thead>
<tr>
<th>Exponent + 1023</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(11 bits)</td>
<td>(52 bits)</td>
</tr>
<tr>
<td>0 1</td>
<td>12 63</td>
</tr>
</tbody>
</table>

Figure 19-5. BFP Long Format (8 bytes)

When a number or NaN in the BFP long format is loaded into a floating-point register, it occupies the entire register.

BFP Extended Format

<table>
<thead>
<tr>
<th>Exponent + 16383</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(15 bits)</td>
<td>(112 bits)</td>
</tr>
<tr>
<td>0 1</td>
<td>16 127</td>
</tr>
</tbody>
</table>

Figure 19-6. BFP Extended Format (16 bytes)

A number or NaN in the BFP extended format occupies a register pair. The sign and biased exponent are in the leftmost 16 bits of the left register and are followed by the leftmost 48 bits of the fraction. The rightmost 64 bits of the fraction are in the right register of the pair.

The properties of the three formats are tabulated in Figure 19-7 on page 19-5.

Biased Exponent

For each format, the bias that is used to allow all exponents to be expressed as unsigned numbers is shown in Figure 19-7 on page 19-5. Biased exponents are similar to the characteristics of the HFP format, except that special meanings are attached to biased exponents of all zeros and all ones, which are discussed in the section “Classes of BFP Data” on page 19-5.

Significand

In each format, the binary point of a BFP number is considered to be to the left of the leftmost fraction bit. To the left of the binary point there is an implied unit bit, which is considered to be one for normalized numbers and zero for zeros and denormalized numbers. The fraction with the implied unit bit appended on the left is the significand of the number.

The value of a normalized BFP number is the significand multiplied by the radix 2 raised to the power of the unbiased exponent. The value of a denormalized BFP number is the significand multiplied by the radix 2 raised to the power of the minimum exponent.

A value of one in the rightmost bit position of the significand in each format is sometimes referred to as one ulp (unit in the last place).

Values of Nonzero Numbers

The values of nonzero numbers in the various formats are shown in Figure 19-8.

<table>
<thead>
<tr>
<th>Number Class</th>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized</td>
<td>Short</td>
<td>±2^{e-127} x (1.f)</td>
</tr>
<tr>
<td></td>
<td>Long</td>
<td>±2^{e-1023} x (1.f)</td>
</tr>
<tr>
<td></td>
<td>Extended</td>
<td>±2^{e-16383} x (1.f)</td>
</tr>
<tr>
<td>Denormalized</td>
<td>Short</td>
<td>±2^{e-126} x (0.f)</td>
</tr>
<tr>
<td></td>
<td>Long</td>
<td>±2^{e-1022} x (0.f)</td>
</tr>
<tr>
<td></td>
<td>Extended</td>
<td>±2^{e-16382} x (0.f)</td>
</tr>
</tbody>
</table>

Explanation:

e  Biased exponent (shown in decimal).
f  Fraction (in binary).

Figure 19-8. Values of Nonzero Numbers

Programming Note: The IEEE standard specifies minimum requirements for the extended
Table 19.1: Summary of BFP Data Formats

<table>
<thead>
<tr>
<th>Property</th>
<th>Format length (bits)</th>
<th>Biased-exponent length (bits)</th>
<th>Fraction length (bits)</th>
<th>Precision (p)</th>
<th>Maximum exponent (Emax)</th>
<th>Minimum exponent (Emin)</th>
<th>Exponent bias</th>
<th>Nmax</th>
<th>Nmin</th>
<th>Dmin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>32</td>
<td>8</td>
<td>23</td>
<td>24</td>
<td>127</td>
<td>-126</td>
<td>127</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long</td>
<td>64</td>
<td>11</td>
<td>52</td>
<td>53</td>
<td>1023</td>
<td>-1022</td>
<td>1023</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended</td>
<td>128</td>
<td>15</td>
<td>112</td>
<td>113</td>
<td>16383</td>
<td>-16382</td>
<td>16383</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:
- Value is approximate.
- Dmin: Smallest (in magnitude) representable denormalized number.
- Nmax: Largest (in magnitude) representable number.
- Nmin: Smallest (in magnitude) representable normalized number.

Figure 19-7. Summary of BFP Data Formats

The BFP extended format meets these requirements, far exceeding them in the area of precision.

**Classes of BFP Data**

There are six classes of BFP data, which include numeric and related nonnumeric entities. Each data item consists of a sign, an exponent, and a significand. The exponent is biased such that all biased exponents are nonnegative unsigned numbers and the minimum biased exponent is zero. The significand consists of an explicit fraction and an implicit unit bit to the left of the binary point. The sign bit is zero for plus and one for minus.

All finite nonzero numbers within the normalized range permitted by a given format have a unique BFP representation. There are no unnormalized numbers, which numbers might allow multiple representations for the same values, and there are no unnormalized arithmetic operations. Tiny numbers of a magnitude below the minimum normalized number in a given format are represented as denormalized numbers, but those values are also represented uniquely. The implied unit bit of a normalized number is one, and that of a denormalized number or a zero is zero.

The six classes of BFP data are summarized in Figure 19-9 on page 19-6.
The instruction TEST DATA CLASS may be used to determine the class of a BFP operand.

**Zeros**

Zeros have a biased exponent of zero and a zero fraction. The implied unit bit is zero. A +0 is distinct from −0, except that comparison treats them as equal.

**Denormalized Numbers**

Denormalized numbers are numbers which are smaller than the smallest normalized number and greater than zero in magnitude. They have a biased exponent of zero and a nonzero fraction. The biased exponent is treated arithmetically as if it were one, which causes the exponent to be the minimum exponent. The implied unit bit is zero.

**Normalized Numbers**

Normalized numbers have a biased exponent greater than zero but less than all ones. The implied unit bit is one, and the fraction may have any value.

**Infinities**

An infinity is represented by a biased exponent of all ones and a zero fraction. Infinities can participate in most arithmetic operations and give a consistent result, usually infinity. In comparisons, +∞ compares greater than any finite number, and −∞ compares less than any finite number.

**Signaling and Quiet NaNs**

A NaN (not-a-number) entity is represented by a biased exponent of all ones and a nonzero fraction. NaNs are produced in place of a numeric result after an invalid operation when there is no interruption. NaNs may also be used by the program to flag special operands, such as the contents of an uninitialized storage area.

There are two types of NaNs, signaling and quiet. A signaling NaN (SNaN) is distinguished from the corresponding quiet NaN (QNaN) by the leftmost fraction bit: zero for the SNaN and one for the QNaN. A special QNaN is supplied as the default result for an IEEE-invalid-operation condition; it has a plus sign and a leftmost fraction bit of one, with the remaining fraction bits being set to zeros. Normally, QNaNs are just propagated during computations so that they will remain visible at the end. An SNaN operand causes an IEEE-invalid-operation exception. If the IEEE-invalid-operation mask (FPC 0.0) is zero, the result is the corresponding QNaN, which is produced by setting the leftmost fraction bit to one, and the IEEE-invalid-operation flag (FPC 1.0) is set to one. If the IEEE-invalid-operation mask (FPC 0.0) is one, the operation is suppressed, and a data exception for IEEE-invalid operation occurs.

**Programming Notes:**

1. The program can generate and assign meanings to any nonzero fraction values of a NaN. The CPU propagates those values unchanged, except that an SNaN is changed to the corresponding QNaN if the IEEE-invalid-operation mask bit is zero, and conversion to a narrower format may truncate significant bits on the right.

2. The standard requires SNaNs to signal the invalid-operation exception for the arithmetic, comparison, and conversion operations that are part of the standard, but it makes it an implementation option whether copying an SNaN without a change of format signals the
exception. In the appendix, the standard also makes it an implementation option whether SNaNs should signal the invalid-operation exception for the recommended functions of copying the sign, taking the absolute value, reversing the sign, and testing the data class of a number.

The above functions generally correspond to the instructions LOAD, LOAD COMPLEMENT, LOAD NEGATIVE, LOAD POSITIVE, and TEST DATA CLASS. These instructions do not signal the invalid-operation exception but, instead, treat SNaNs like any other data; giving an exception would be disruptive when the intention is to include SNaNs. TEST DATA CLASS does not give an exception since it is the instruction with which to test for the presence of SNaNs.

3. LOAD AND TEST signals the invalid-operation exception when the operand is an SNaN. This instruction, in conjunction with the above instructions, gives the program the choice of either option permitted by the standard.

4. Load-type instructions which change the precision signal the invalid-operation exception when the operand is an SNaN, as this is required by the standard.

BFP-Format Conversion

The instructions LOAD LENGTHENED and LOAD ROUNDED perform conversions of numbers between the short, long, and extended formats. For BFP formats, conversion involves adjustments to both the fraction and the exponent. When converting a normalized number to a wider format (short to long, long to extended, or short to extended), the fraction is adjusted by appending sufficient zeros on the right. Conversion to a narrower format requires rounding of the fraction before dropping excess bits on the right, and an IEEE-inexact condition may result.

The exponent is adjusted by adding or subtracting the difference in the biases of the two formats. When converting to a narrower format, this adjustment causes IEEE underflow if the resultant biased exponent would be less than one, or IEEE overflow if the resultant exponent would be equal to or greater than the maximum exponent for the new format.

When a denormalized number is converted to a wider format, the biased exponent of the source operand is treated as if it had the value one. The result is normalized.

Programming Notes:

1. When a NaN is converted to a narrower format, the appropriate number of fraction bits on the right are simply dropped with no indication. This is unlike the conversion of nonzero numbers, where the loss of nonzero fraction bits causes an IEEE-inexact condition. Thus, programs which encode NaN fraction bits for specific purposes must ensure that the distinguishing bits are placed in the left part of the fraction.

2. Converting a NaN from a wide format to a narrower format cannot turn the NaN into an infinity because an SNaN either causes an interruption or turns into a QNaN, and all QNaNs have a leftmost fraction bit of one.

BFP Rounding

Arithmetic and conversion operations are performed as if they first produced an intermediate result correct to infinite precision and with unbounded range. If this intermediate result can be represented exactly in the target format, then it is given exactly. Otherwise, the intermediate result is replaced by one of the two closest values that can be represented, the choice depending on the rounding mode.

Rounding is performed automatically as part of every arithmetic and conversion operation. The precision of the target (short, long, or extended) is specified by the operation code.

Rounding Mode

There are four rounding modes. The current rounding mode is specified by the value of two rounding-mode bits in the FPC register, as follows:

00 Round to nearest (default). Round the intermediate result up or down to the nearest representable value; that is, add, ignoring the sign, a one to the bit just beyond the last result bit to be retained, propagate the carry, and discard the bits beyond the last one to be retained. If the difference was exactly one-half ULP (a one in the bit position just beyond the last place, with all zeros beyond
that), the nearest even number is chosen; that is, after the rounding addition, the last result bit retained is set to zero.

If the absolute value of the intermediate result is equal to or greater than the largest representable number plus one-half ulp, that is, if the absolute value is equal to or greater than $2^{E_{\text{max}}} \times (2 - 2^{-p})$, the result is rounded to infinity with the same sign as the intermediate result.

01 Round toward 0. Discard all bits to the right of the last intermediate-result bit to be retained.

10 Round toward $+\infty$. If the intermediate result is positive and there are any ones to the right of the last result bit to be retained, add one to that bit. Then, for either sign, discard the bits beyond the last one to be retained.

11 Round toward $-\infty$. If the intermediate result is negative and there are any ones to the right of the last result bit to be retained, subtract one from that bit (that is, add one to the magnitude). Then, for either sign, discard the bits beyond the last one to be retained.

Programming Notes:
1. Rounding a finite result toward zero cannot give infinity.
2. Rounding a result toward $+\infty$ can give $+\infty$ but not $-\infty$.
3. Rounding a result toward $-\infty$ can give $-\infty$ but not $+\infty$.

Normalization and Denormalization

Every arithmetic or conversion operation is considered to produce an intermediate result as if the precision and exponent range were unbounded, unless the result is defined to be zero, infinity, or NaN. The final result is produced by normalizing and then rounding this intermediate result. When there is exponent underflow, that is, the biased exponent of the normalized intermediate result is less than one, then the intermediate result is denormalized to produce the final result, as described below.

Denormalization consists in shifting the significand, including the units bit, to the right while introducing zero bits on the left, and in increasing the exponent by one for each bit of shift. When the biased exponent reaches +1, the significand is rounded according to the current rounding mode. If all bits of the rounded significand are zeros, the result is made zero. If rounding produces a carry into the units bit position of the significand, the biased exponent remains +1, since this result is a normalized number ($\pm 2^{E_{\text{min}}}$). Otherwise, the units bit remains zero, the biased exponent is set to zero, and the result is considered denormalized.

Arithmetic operations on denormalized operands are performed as if the operands had first been normalized.

Intermediate results are first normalized or denormalized, as required, and then rounded. This avoids double rounding of a single operation, which might increase the rounding error. (Any right shift required after a carry from rounding to renormalize the result does not require a second rounding, because the bit shifted off on the right is always zero.)

**BFP Comparison**

Comparisons are always exact and cannot cause an IEEE-inexact condition.

Comparison ignores the sign of zero, that is, $+0$ equals $-0$.

Infinities with like sign compare equal, that is, $+\infty$ equals $+\infty$, and $-\infty$ equals $-\infty$.

A NaN compares as unordered with any other operand, whether a finite number, an infinity, or another NaN, including itself.

Two sets of instructions are provided: COMPARE and COMPARE AND SIGNAL. In the absence of QNaNs, these instructions work the same. These instructions work differently only when both of the following are true:

- Neither operand of the instruction is an SNaN
- At least one operand of the instruction is a QNaN

In this case, COMPARE simply sets condition code 3, but COMPARE AND SIGNAL recognizes the IEEE-invalid-operation condition. If any
operand is an SNaN, both instructions recognize
the IEEE-invalid-operation condition.

The action when the IEEE-invalid-operation condi-
tion is recognized depends on the
IEEE-invalid-operation mask bit in the FPC reg-
ister. If the mask bit is zero, then the instruction
execution is completed by setting condition code
3, and the IEEE-invalid-operation flag in the FPC
register is set to one. If the mask bit is one, then
the condition is reported as a program interruption
for a data exception with DXC 80 hex (IEEE
invalid operation).

**Programming Note:** A compiler can select either
COMPARE or COMPARE AND SIGNAL for a
comparison, depending on whether the IEEE
standard or a relevant language standard requires
a QNaN to be recognized as an exception condi-
tion.

### Condition Codes for BFP Instructions

For arithmetic operations with finite or infinite
numeric results, condition codes 0, 1, and 2 are
set to indicate that the result is a zero of either
sign, less than zero, or greater than zero, respec-
tively. The condition-code setting depends only
on an inspection of the rounded result. For com-
parison operations, condition codes 0, 1, and 2
indicate equal, low, or high, respectively. These
settings are the same as for the HFP instructions.

Condition code 3 can also be set. After an arith-
metic operation, condition code 3 indicates a NaN
result of either sign. After a comparison, it indi-
cates that a NaN was involved in the comparison
(the unordered condition). See Figure 19-10.

<table>
<thead>
<tr>
<th>CC</th>
<th>Arithmetic</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>±0</td>
<td>Equal</td>
</tr>
<tr>
<td>1</td>
<td>&lt;0</td>
<td>Low</td>
</tr>
<tr>
<td>2</td>
<td>&gt;0</td>
<td>High</td>
</tr>
<tr>
<td>3</td>
<td>±NaN</td>
<td>Unordered</td>
</tr>
</tbody>
</table>

*Figure 19-10. Condition Codes*

### Remainder

The instruction DIVIDE TO INTEGER produces
two floating-point results, an exact integer quotient
and the corresponding remainder. The remainder
is defined as follows:

Let

\[ a = \text{Dividend} \]
\[ b = \text{Divisor} \]
\[ q = \text{Exact quotient } (a/b) \]
\[ r = \text{Remainder} \]

in the selected floating-point format. Then

\[ r = a - b \cdot n \]

where \( n \) is an integer. If \( q \) is an integer, then \( n \)
equals \( q \). Otherwise, \( n \) is obtained by rounding \( q \)
according to a specified quotient rounding mode.

When the specified quotient rounding mode is
round to nearest or round toward zero, the
remainder is exact for any finite dividend and any
nonzero divisor. The remainder cannot overflow.

If the integer quotient has a value that lies outside
the range of the operand format, a wrapped result
is provided.

In certain cases where the number of bits in the
integer quotient exceeds or may exceed the
maximum number of bits provided in the precision
of the operand format, partial results are
produced, and more than one execution of the
instruction is required to obtain the final result; this
may be done with a simple instruction loop.

Partial results are produced when the precise quo-
tient is not an integer and the two integers closest
to this precise quotient cannot both be repres-
ented exactly in the precision of the quotient. This
situation exists when the precise quotient is
greater than \( 2^p \), where \( P \) is the precision of the
operand format, and the remainder is not zero.
When the remainder is zero, then the quotient is
an integer, and the number of bits required to rep-
resent the quotient is never more than the preci-
sion of the target.

**Programming Note:** The remainder result of
DIVIDE TO INTEGER with a specified quotient
rounding mode of round to nearest corresponds to
the *Remainder* function in the IEEE standard.
This function is similar to the *MOD* function found
in some languages and to the mathematical
modulo function, but they are not the same. They differ in the definition of n:

- Remainder: n is q rounded to nearest.
- Modulo: n is q rounded toward \(-\infty\).
- MOD: n is q rounded toward 0.

Another important difference is that implementations of modulo and MOD may put range restrictions on the result because they may simply use the DIVIDE instruction and accept its range restrictions.

The MOD definition provides an exact result, as does Remainder, but the modulo definition may result in rounding errors.

The differences between the various methods may be illustrated by the simple example of computing a divided by b to obtain an integer quotient n, where a is a series of integers, and b is +4 or −4. Figure 19-11 on page 19-11 shows the results for the three definitions.

The result of Remainder lies in the range of zero to one-half the divisor, inclusive, in magnitude. A zero result is defined to have the sign of the dividend. A zero divisor is invalid.

The modulo and MOD results can both be computed from the Remainder result; the reverse may not be true, because of rounding errors and, depending on the implementation, range restrictions.

An extreme example of the rounding error that can occur with the modulo definition is the following, where the result is restricted to two significant decimal digits:

- modulo(0.01,-95) = −94.99, which rounds to −95
- Remainder(0.01,-95) = 0.01

The properly rounded modulo result is completely wrong since it is equal to the divisor instead of being smaller in magnitude. The Remainder result is exact and can be used to compute the theoretical result of modulo.

Remainder is included as an arithmetic operation because of its usefulness in argument reduction when computing elementary transcendental functions. Thus, SIN(X) can be computed to full precision for any value of X in degrees by first reducing the argument to Remainder(X,360).

IEEE Exception Conditions

The results of each of the IEEE exception conditions are controlled by a mask bit in the FPC register. When an IEEE exception condition is recognized, one of two actions is taken:

- If the corresponding mask bit in the FPC register is zero, a default action is taken, as specified for each condition, and the corresponding flag bit in the FPC register is set to one. Program execution then continues normally.
- If the corresponding mask bit in the FPC register is one, a program interruption for a data exception occurs, the operation is suppressed or completed, depending on the condition, and the data-exception code (DXC) assigned for that condition is provided.

IEEE Invalid Operation

An IEEE-invalid-operation condition is recognized when, in the execution of a BFP instruction, any of the following occurs:

1. An SNaN is encountered in any BFP arithmetic, comparison, or conversion operation or by LOAD AND TEST.
2. A QNaN is encountered in a BFP comparison by COMPARE AND SIGNAL.
3. A BFP difference is undefined (addition of infinities of opposite sign, or subtraction of infinities of like sign).
4. A BFP product is undefined (zero times infinity).
5. A BFP quotient is undefined (DIVIDE instruction with both operands zero or both operands infinity).
6. A BFP remainder is undefined (DIVIDE TO INTEGER with a dividend of infinity or a divisor of zero).
7. A BFP square root is undefined (negative nonzero operand).

If the IEEE-invalid-operation mask bit in the FPC register is zero, the IEEE-invalid-operation flag bit in the FPC register is set to one. The completion of the operation depends on the type of operation and the operands.
If the instruction performs a comparison and no program interruption occurs, the comparison result is unordered.

If the instruction is one that produces a BFP result, if no program interruption occurs, and if none of the operands is a NaN, the result is the default QNaN. If one of the operands is a NaN, that operand becomes the result unchanged, except that an SNaN is first converted to the corresponding QNaN by setting the leftmost fraction bit to one.

If the IEEE-invalid-operation mask bit in the FPC register is one, the operation is suppressed, and the condition is reported as a program interruption for a data exception with DXC 80 hex.

IEEE Division-By-Zero
An IEEE-division-by-zero condition is recognized when in BFP division the divisor is zero and the dividend is a finite nonzero number.

If the IEEE-division-by-zero mask bit in the FPC register is zero, the IEEE-division-by-zero flag bit in the FPC register is set to one. The operation is completed using as the result an infinity with a sign that is the exclusive or of the dividend and divisor signs.

If the IEEE-division-by-zero mask bit in the FPC register is one, the operation is suppressed, and the condition is reported as a program interruption for a data exception with DXC 40 hex.

IEEE Overflow
An IEEE-overflow condition is recognized when the exponent of the rounded result of a BFP operation would be greater than the maximum exponent of the target format if the exponent range were unbounded.

If the IEEE-overflow mask bit in the FPC register is zero, the IEEE-overflow flag bit in the FPC register is set to one. The result of the operation depends on the sign of the intermediate result and on the current rounding mode:

1. When rounding to nearest, the result is infinity with the sign of the intermediate result.
2. When rounding toward 0, the result is the largest finite number of the format, with the sign of the intermediate result.
3. When rounding toward $+\infty$, the result is $+\infty$ if the sign is plus, or it is the finite negative number with the largest magnitude if the sign is minus.

4. When rounding toward $-\infty$, the result is the largest finite positive number if the sign is plus or $-\infty$ if the sign is minus.

If the IEEE-overflow mask bit in the FPC register is one, the operation is completed by producing a wrapped result, and the condition is reported as a program interruption for a data exception with DXC 20, 28, or 2C hex, depending on whether the wrapped result is exact, inexact and truncated, or inexact and incremented, respectively.

**IEEE Underflow**

An IEEE-underflow condition is recognized when the exponent of the exact result of a BFP operation would be less than the minimum exponent of the target format.

If the IEEE-underflow mask bit in the FPC register is zero, then the action depends on whether the result can be represented exactly and, if not, also on the setting of the IEEE-inexact mask bit in the FPC register. If the result can be represented exactly, the operation is completed by denormalizing the intermediate result. If the result cannot be represented exactly and the IEEE-inexact mask bit in the FPC register is zero, the operation is completed by denormalizing and rounding the intermediate result, and the IEEE-underflow and IEEE-inexact flag bits in the FPC register are set to ones. If the result cannot be represented exactly and the IEEE-inexact mask bit in the FPC register is one, the IEEE-inexact flag bit in the FPC register is set to one.

If the IEEE-underflow mask bit in the FPC register is one, the operation is completed using the rounded result or, in case of overflow or underflow, the result specified for IEEE overflow or IEEE underflow.

If the IEEE-inexact mask bit in the FPC register is zero, the IEEE-inexact flag bit in the FPC register is set to one.

If the IEEE-inexact mask bit in the FPC register is one, the operation is completed, and the condition is reported as a program interruption for a data exception with DXC 08 or 0C hex, depending on whether the result is inexact and truncated or inexact and incremented, respectively.

**Programming Notes:**

1. All IEEE traps are reported by means of a program interruption for a data exception with a data-exception code. The use of data exception provides the application program with a convenient interface since this exception is one of the original 15 exceptions in the System/360 architecture and is supported by most control programs that support the ESA/390 architecture.

2. The IEEE standard includes recommendations for the trap handler. When a system traps, the trap handler should be able to determine:
   
a. Which exception(s) occurred on this operation.
   
b. The kind of operation that was being performed.
   
c. The destination’s format.
   
d. For overflow, underflow, and inexact exceptions, the correctly rounded result, including information that might not fit in the destination’s format.
   
e. For invalid-operation and divide-by-zero exceptions, the operand values.

Items a and d are supplied as part of the interruption action. Items b, c, and e can be obtained starting with the instruction address.
in the old PSW and from this finding the instruction (which indicates the operation and format) and then the operands.

3. The description of underflow is one of the most difficult parts of the standard to understand. This is because:
   a. The condition is described as two “correlated events” — “tininess” and “loss of accuracy.”
   b. For tininess, the standard provides two options for detection: “after rounding” or “before rounding.”
   c. For loss of accuracy, the standard provides two options for detection: “denormalization loss” or “inexact result.”
   d. Implementation of the trap is optional.
   e. The conditions to signal underflow are different depending on whether or not the trap is taken.

Each of the above items is discussed below.

   a. Tininess refers to a nonzero number strictly between $\pm 2^{E_{\text{min}}}$. (All denormalized numbers are in this range.) Loss of accuracy means that the result cannot be represented exactly.
   b. Detection of tininess after or before rounding differs only for the case when “rounding” would increase the magnitude of the result to exactly $\pm 2^{E_{\text{min}}}$. It must be noted, however, that the action which the standard here calls “rounding” is not the rounding to produce the delivered result but rounding to compute an intermediate value having the precision of the result but “as though the exponent range were unbounded.” In fact, it is possible that the delivered result may not be tiny even though the intermediate value “after rounding” is tiny.

   The option selected in the ESA/390 BFP architecture (and the RS/6000) is to detect tininess before rounding.

   c. The difference between detection of loss of accuracy as a denormalization loss or as an inexact result can best be understood by considering two intermediate values: (1) a precise intermediate value, which has unbounded precision and unbounded exponent range, and (2) a rounded intermediate value, which is obtained by rounding the precise intermediate value to the precision of the result but with unbounded exponent range. Inexact result is said to occur when the delivered result differs from the precise intermediate value. Denormalization loss is said to occur when the delivered result differs from the rounded intermediate value. The two options differ in the case when the delivered result is equal in value to the rounded intermediate value but these are not equal to the precise intermediate value. Although the standard uses the term “denormalization loss,” this condition includes a case in which the delivered result is normalized.

   The option selected in the ESA/390 BFP architecture (and the RS/6000) is to detect “loss of accuracy” as an inexact result.

   d. Although the standard does not require traps to be implemented for underflow or the other arithmetic exceptions, it does state that “with each exception should be associated a trap under user control.” Since it also defines “should” as “that which is strongly recommended as being in keeping with the intent of the standard,” the ESA/390 BFP architecture provides traps by means of program interruptions.

   e. When the underflow trap is enabled, underflow is to be signaled when tininess is detected regardless of loss of accuracy. When the underflow trap is not enabled, the underflow flag bit is to be set only when both tininess and loss of accuracy have been detected. Add and subtract can result in tiny or inexact results, but not both. Thus, when underflow is disabled, add and subtract never set the underflow flag bit.

---

Result Figures

Concise descriptions of the results produced by many of the BFP instructions are made by means of figures which contain columns and rows representing all possible combinations of BFP data class for the source operands of an instruction. The information shown at the intersection of a row and a column is one or more symbols representing the result or results produced for that particular combi-
nation of source-operand data classes. Explanations of the symbols used are contained in each figure. In many cases, the explanation of a particular result is in the form of a cross reference to another figure. In many cases, the information shown at the intersection consists of several symbols separated by commas. All such results are produced unless one of the results is a program interruption. In the case of a program interruption, the operation is suppressed or completed as shown in Figure 19-13.

**Data-Exception Codes (DXC) and Abbreviations**

Figure 19-12 on page 19-15 shows IEEE exception-condition and flag abbreviations that are used in the result figures, and it explains the symbols “Xi:” and “Xz:” that are used in the figures. Bits 0-4 (i,z,o,u,x) of the eight-bit data exception code (DXC) in byte 2 of the FPC register are trap flags and correspond to the same bits in bytes 0 and 1 of the register (IEEE masks and IEEE flags). The trap flag for an exception, instead of the IEEE flag, is set to one when an interruption for the exception is enabled by the corresponding IEEE mask bit. Bit 5 of byte 2 (y) is used in conjunction with bit 4, inexact (x), to indicate that the result has been incremented in magnitude.

Figure 19-13 shows the various DXCs that can be indicated, the associated instruction endings, and abbreviations that are used for the DXCs in the result figures. (The abbreviation “PID” stands for “program interruption for a data exception.”)

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>DXC (Hex)</th>
<th>Data-Exception-Code Name</th>
<th>Instruction Ending</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIDx</td>
<td>08</td>
<td>IEEE inexact and truncated</td>
<td>Complete</td>
</tr>
<tr>
<td>PIDy</td>
<td>0C</td>
<td>IEEE inexact and incremented</td>
<td>Complete</td>
</tr>
<tr>
<td>PIDu</td>
<td>10</td>
<td>IEEE underflow, exact</td>
<td>Complete, wrap exponent</td>
</tr>
<tr>
<td>PIDux</td>
<td>18</td>
<td>IEEE underflow, inexact and truncated</td>
<td>Complete, wrap exponent</td>
</tr>
<tr>
<td>PIDuy</td>
<td>1C</td>
<td>IEEE underflow, inexact and incremented</td>
<td>Complete, wrap exponent</td>
</tr>
<tr>
<td>PIDo</td>
<td>20</td>
<td>IEEE overflow, exact</td>
<td>Complete, wrap exponent</td>
</tr>
<tr>
<td>PIDox</td>
<td>28</td>
<td>IEEE overflow, inexact and truncated</td>
<td>Complete, wrap exponent</td>
</tr>
<tr>
<td>PIDoy</td>
<td>2C</td>
<td>IEEE overflow, inexact and incremented</td>
<td>Complete, wrap exponent</td>
</tr>
<tr>
<td>PIDz</td>
<td>40</td>
<td>IEEE division by zero</td>
<td>Suppress</td>
</tr>
<tr>
<td>PIDi</td>
<td>80</td>
<td>IEEE invalid operation</td>
<td>Suppress</td>
</tr>
</tbody>
</table>

**Instructions**

The BFP instructions and their mnemonics and operation codes are listed in Figure 19-14 on page 19-16. The figure indicates, in the column labeled “Characteristics,” the instruction format, when the condition code is set, the instruction fields that designate access registers, and the exceptional conditions in operand designations, data, or results that cause a program interruption.

All BFP instructions are subject to the AFP-register-control bit, bit 45 of control register 0. For the BFP instructions to be executed successfully, the AFP-register-control bit must be one; otherwise, a BFP-instruction data exception, DXC 2, is recognized.

Mnemonics for the BFP instructions are distinguished from the corresponding HFP instructions by a B in the mnemonic. Mnemonics for the BFP instructions have an R as the last letter when the instruction is in the RRE or RRF format. Certain letters are used for BFP instructions to represent operand-format length, as follows:

- F Thirty-two-bit fixed point
- G Sixty-four-bit fixed point
- D Long
<table>
<thead>
<tr>
<th>Exception Condition</th>
<th>FPC Mask Bit</th>
<th>IEEE Flag</th>
<th>Abbr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE invalid operation</td>
<td>Xi&lt;sup&gt;1&lt;/sup&gt;</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>IEEE division by zero</td>
<td>Xz&lt;sup&gt;2&lt;/sup&gt;</td>
<td>0.1</td>
<td>1.1</td>
</tr>
<tr>
<td>IEEE overflow</td>
<td>Xo</td>
<td>0.2</td>
<td>1.2</td>
</tr>
<tr>
<td>IEEE underflow</td>
<td>Xu</td>
<td>0.3</td>
<td>1.3</td>
</tr>
<tr>
<td>IEEE inexact</td>
<td>Xx</td>
<td>0.4</td>
<td>1.4</td>
</tr>
</tbody>
</table>

**Explanation:**

1. The symbol “Xi:” followed by a list of results in a figure indicates that, when FPC 0.0 is zero, then instruction execution is completed by setting SFi (FPC 1.0) to one and producing the indicated results; and when FPC 0.0 is one, then instruction execution is suppressed, the data exception code (DXC) is set to 80 hex, and a program interruption for a data exception occurs.

2. The symbol “Xz:” followed by a list of results in a figure indicates that, when FPC 0.1 is zero, then instruction execution is completed by setting SFz (FPC 1.1) to one and producing the indicated results; and when FPC 0.1 is one, then instruction execution is suppressed, the data exception code (DXC) is set to 40 hex, and a program interruption for a data exception occurs.

---

**Note:** In the detailed descriptions of the individual instructions, the mnemonic and the symbolic operand designation for the assembler language are shown with each instruction. For a register-to-register operation using COMPARE (short), for example, CEBR is the mnemonic and R<sub>1</sub>, R<sub>2</sub> the operand designation.
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (extended BFP)</td>
<td>AXBR</td>
<td>RRE C</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ADD (long BFP)</td>
<td>ADBR</td>
<td>RRE C</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ADD (long BFP)</td>
<td>ADB</td>
<td>RXE C A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ADD (short BFP)</td>
<td>AEBR</td>
<td>RRE C</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ADD (short BFP)</td>
<td>AEB</td>
<td>RXE C A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>COMPARE (extended BFP)</td>
<td>CXBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE (long BFP)</td>
<td>CDBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE (long BFP)</td>
<td>CDB</td>
<td>RXE C A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE (short BFP)</td>
<td>CEBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE (short BFP)</td>
<td>CEB</td>
<td>RXE C A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (extended BFP)</td>
<td>KXBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (long BFP)</td>
<td>KDBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (long BFP)</td>
<td>KDB</td>
<td>RXE C A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (short BFP)</td>
<td>KEBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (short BFP)</td>
<td>KEB</td>
<td>RXE C A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to ext. BFP)</td>
<td>CXFBR</td>
<td>RRE N</td>
<td>Db</td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to long BFP)</td>
<td>CDFBR</td>
<td>RRE N</td>
<td>Db</td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to short BFP)</td>
<td>CEFBR</td>
<td>RRE N</td>
<td>Db Xx</td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to ext. BFP)</td>
<td>CXGBR</td>
<td>RRE N</td>
<td>Db</td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to long BFP)</td>
<td>CDGBR</td>
<td>RRE N</td>
<td>Db Xx</td>
</tr>
<tr>
<td>CONVERT TO FIXED (ext. BFP to 32)</td>
<td>CFXBR</td>
<td>RRE C</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>CONVERT TO FIXED (long BFP to 32)</td>
<td>CFDBR</td>
<td>RRE C</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>CONVERT TO FIXED (short BFP to 32)</td>
<td>CFEBR</td>
<td>RRE C</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>CONVERT TO FIXED (ext. BFP to 64)</td>
<td>CGXBR</td>
<td>RRE C N</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>CONVERT TO FIXED (long BFP to 64)</td>
<td>CGDBR</td>
<td>RRE C N</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>DIVIDE (extended BFP)</td>
<td>DXBR</td>
<td>RRE N</td>
<td>Db Xi Xz Xo Xu Xx</td>
</tr>
<tr>
<td>DIVIDE (long BFP)</td>
<td>DDBR</td>
<td>RRE N</td>
<td>Db Xi Xz Xo Xu Xx</td>
</tr>
<tr>
<td>DIVIDE (long BFP)</td>
<td>DDB</td>
<td>RXE A</td>
<td>Db Xi Xz Xo Xu Xx</td>
</tr>
<tr>
<td>DIVIDE (short BFP)</td>
<td>DEBR</td>
<td>RRE A</td>
<td>Db Xi Xz Xo Xu Xx</td>
</tr>
<tr>
<td>DIVIDE (short BFP)</td>
<td>DEB</td>
<td>RXE A</td>
<td>Db Xi Xz Xo Xu Xx</td>
</tr>
<tr>
<td>DIVIDE TO INTEGER (long BFP)</td>
<td>DIDBR</td>
<td>RRE C</td>
<td>Db Xi Xz Xo Xu Xx</td>
</tr>
<tr>
<td>DIVIDE TO INTEGER (short BFP)</td>
<td>DIEBR</td>
<td>RRE C</td>
<td>Db Xi Xz Xo Xu Xx</td>
</tr>
<tr>
<td>EXTRACT FPC</td>
<td>EFPC</td>
<td>RRE</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD AND TEST (extended BFP)</td>
<td>LTXBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD AND TEST (long BFP)</td>
<td>LTDBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD AND TEST (short BFP)</td>
<td>LTEBR</td>
<td>RRE C</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (extended BFP)</td>
<td>LCXBR</td>
<td>RRE C</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (long BFP)</td>
<td>LDBR</td>
<td>RRE C</td>
<td>Db</td>
</tr>
</tbody>
</table>

Figure 19-14 (Part 1 of 3). Summary of BFP Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD COMPLEMENT (short BFP)</td>
<td>LCEBR</td>
<td>RRE C</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD FP INTEGER (extended BFP)</td>
<td>FIXBR</td>
<td>RRF SP</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>LOAD FP INTEGER (long BFP)</td>
<td>FIDBR</td>
<td>RRF SP</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>LOAD FP INTEGER (short BFP)</td>
<td>FIEBR</td>
<td>RRF SP</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>LOAD FPC</td>
<td>LFPC S</td>
<td>A SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD LENGTHENED (long to ext. BFP)</td>
<td>LXDBR</td>
<td>RRE SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (long to ext. BFP)</td>
<td>LXDB</td>
<td>RXE A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. BFP)</td>
<td>LXEBR</td>
<td>RRE SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. BFP)</td>
<td>LXEB</td>
<td>RXE A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>LDEBR</td>
<td>RRE SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>LDEB</td>
<td>RXE A SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD NEGATIVE (extended BFP)</td>
<td>LNXBR</td>
<td>RRE C SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD NEGATIVE (long BFP)</td>
<td>LNDBR</td>
<td>RRE C</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD NEGATIVE (short BFP)</td>
<td>LNEBR</td>
<td>RRE C SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD POSITIVE (extended BFP)</td>
<td>LPXBR</td>
<td>RRE C SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD POSITIVE (long BFP)</td>
<td>LPDBR</td>
<td>RRE C</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD POSITIVE (short BFP)</td>
<td>LPDB</td>
<td>RXE A SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to long BFP)</td>
<td>LDBR</td>
<td>RRE SP</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to short BFP)</td>
<td>LEBR</td>
<td>RRE SP</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>LOAD ROUNDED (long to short BFP)</td>
<td>LEBR</td>
<td>RRE</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY (extended BFP)</td>
<td>MXBR</td>
<td>RRE SP</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY (long BFP)</td>
<td>MDBR</td>
<td>RRE</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY (long to extended BFP)</td>
<td>MXDBR</td>
<td>RRE A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>MULTIPLY (long to extended BFP)</td>
<td>MXDB</td>
<td>RXF A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>MULTIPLY (short BFP)</td>
<td>MEEBR</td>
<td>RRE A SP</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY (short BFP)</td>
<td>MEEB</td>
<td>RXE A SP</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY (short to long BFP)</td>
<td>MEBR</td>
<td>RXE A SP</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY (short to long BFP)</td>
<td>MEB</td>
<td>RXF A SP</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>MADBR</td>
<td>RXF A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>MAEB</td>
<td>RXF A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>MAEB</td>
<td>RXF A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (long BFP)</td>
<td>MSDBR</td>
<td>RXF A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSDB</td>
<td>RXF A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSEBR</td>
<td>RXF A</td>
<td>Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>SET FPC</td>
<td>SFPC RRE</td>
<td>SP</td>
<td>Db</td>
</tr>
<tr>
<td>SET Rounding Mode</td>
<td>SRNM S</td>
<td></td>
<td>Db</td>
</tr>
<tr>
<td>SQUARE ROOT (extended BFP)</td>
<td>SQXBR RRE</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
</tbody>
</table>

Figure 19-14 (Part 2 of 3). Summary of BFP Instructions
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQUARE ROOT (long BFP)</td>
<td>SQDBR</td>
<td>RRE</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>SQUARE ROOT (long BFP)</td>
<td>SQDB</td>
<td>RXE</td>
<td>A</td>
</tr>
<tr>
<td>SQUARE ROOT (short BFP)</td>
<td>SQEBR</td>
<td>RRE</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>SQUARE ROOT (short BFP)</td>
<td>SQEB</td>
<td>RXE</td>
<td>A</td>
</tr>
<tr>
<td>STORE FPC</td>
<td>STFPC</td>
<td>S</td>
<td>DB</td>
</tr>
<tr>
<td>SUBTRACT (extended BFP)</td>
<td>SXBR</td>
<td>RRE C</td>
<td>SP</td>
</tr>
<tr>
<td>SUBTRACT (long BFP)</td>
<td>SDBR</td>
<td>RRE C</td>
<td>A</td>
</tr>
<tr>
<td>SUBTRACT (long BFP)</td>
<td>SDB</td>
<td>RXE C</td>
<td>A</td>
</tr>
<tr>
<td>SUBTRACT (short BFP)</td>
<td>SEBR</td>
<td>RRE C</td>
<td>A</td>
</tr>
<tr>
<td>SUBTRACT (short BFP)</td>
<td>SEB</td>
<td>RXE C</td>
<td>A</td>
</tr>
<tr>
<td>TEST DATA CLASS (extended BFP)</td>
<td>TCXB</td>
<td>RXE C</td>
<td>SP</td>
</tr>
<tr>
<td>TEST DATA CLASS (long BFP)</td>
<td>TCDB</td>
<td>RXE C</td>
<td>SP</td>
</tr>
<tr>
<td>TEST DATA CLASS (short BFP)</td>
<td>TCEB</td>
<td>RXE C</td>
<td>SP</td>
</tr>
</tbody>
</table>

**Explanation:**

- **A** Access exceptions for logical addresses.
- **B2** B2 field designates an access register in the access-register mode.
- **C** Condition code is set.
- **Db** BFP-instruction data exception.
- **N** Instruction is new in z/Architecture as compared to ESA/390.
- **RRE** RRE instruction format.
- **RRF** RRF instruction format.
- **RXE** RXE instruction format.
- **RXF** RXF instruction format.
- **SP** Specification exception.
- **ST** PER storage-alteration event.
- **Xi** IEEE invalid-operation condition.
- **Xo** IEEE overflow condition.
- **Xu** IEEE underflow condition.
- **Xx** IEEE inexact condition.
- **Xz** IEEE division-by-zero condition.

**Figure 19-14 (Part 3 of 3). Summary of BFP Instructions**

**ADD**

Mnemonic1  R1,R2  [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>//////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

**Mnemonic2  R1,D2(X2,B2)  [RXE]**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

**Mnemonic2  Op Code  Operands**

- **AEB**  'B30A' Short BFP
- **ADB**  'B31A' Long BFP
- **AXB**  'B34A' Extended BFP
The second operand is added to the first operand, and the sum is placed at the first-operand location.

If both operands are numeric and finite, they are added algebraically, forming an intermediate sum. The intermediate sum, if nonzero, is normalized and rounded to the operand format according to the current rounding mode. The sum is then placed at the result location.

The sign of the sum is determined by the rules of algebra. This also applies to a result of zero:

- If the result of rounding a nonzero intermediate sum is zero, the sign of the zero result is the sign of the intermediate sum.
- If the sum of two operands with opposite signs is exactly zero, the sign of the result is plus in all rounding modes except round toward \(-\infty\), in which mode the sign is minus.
- The sign of the sum \(x \pm x\) is the sign of \(x\), even when \(x\) is zero.

If one operand is an infinity and the other is finite and numeric, the result is that infinity. If both operands are infinities of the same sign, the result is the same infinity. If the two operands are infinities of opposite signs, an IEEE-invalid-operation condition is recognized.

See [Figure 19-16 on page 19-20](#) for a detailed description of the results of this instruction. (Figure 19-15 is referred to by Figure 19-16.)

For AXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

<table>
<thead>
<tr>
<th>Value of Result (r)</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r=0)</td>
<td>cc0</td>
</tr>
<tr>
<td>(r&lt;0)</td>
<td>cc1</td>
</tr>
<tr>
<td>(r&gt;0)</td>
<td>cc2</td>
</tr>
</tbody>
</table>

**Explanation:**

ccn Condition code is set to n.

See [Figure 19-15. Condition Code for Resultant Sum](#) for a detailed description of the results of this instruction. (Figure 19-15 is referred to by Figure 19-16.)

**Program Exceptions:**

- Access (fetch, operand 2 of AEB and ADB only)
- Data with D XC 2, BFP instruction
- Data with D XC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (AXBR only)

**Programming Note:** Interchanging the two operands in a BFP addition does not affect the value of the sum when the result is numeric. This is not true, however, when both operands are QNaNs, in which case the result is the first operand; or when both operands are SNaNs and the IEEE-invalid-operation mask bit in the FPC register is zero, in which case the result is the QNaN derived from the first operand.
<table>
<thead>
<tr>
<th>First Operand (a)</th>
<th>Results for ADD (a+b) when Second Operand (b) Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>T(−∞), cc1</td>
</tr>
<tr>
<td>−Nn</td>
<td>T(−∞), cc1</td>
</tr>
<tr>
<td>−Dn</td>
<td>T(−∞), cc1</td>
</tr>
<tr>
<td>0</td>
<td>T(−∞), cc1</td>
</tr>
<tr>
<td>+0</td>
<td>T(−∞), cc1</td>
</tr>
<tr>
<td>+Dn</td>
<td>T(−∞), cc1</td>
</tr>
<tr>
<td>+Nn</td>
<td>T(−∞), cc1</td>
</tr>
<tr>
<td>+∞</td>
<td>T(+∞), cc2</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(+∞), cc2</td>
</tr>
<tr>
<td>SNaN</td>
<td>T(+∞), cc2</td>
</tr>
</tbody>
</table>

### Explanation:

- The SNaN is converted to the corresponding QNaN before it is placed at the target operand location.
- ccn Condition code is set to n.
- ccrs Condition code is set according to the resultant sum. See Figure 19-15 on page 19-19.
- dNaN Default quiet NaN.
- Dn Denormalized number.
- Nn Normalized nonzero number.
- R(v) Rounding and range action is performed on the value v. See Figure 19-17 on page 19-21.
- Rezd Exact zero-difference result. See Figure 19-17 on page 19-21.
- T(x) The value x is placed at the target operand location.
- Xi: IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.

**Figure 19-16. Results: ADD**
<table>
<thead>
<tr>
<th>Range of v</th>
<th>Case</th>
<th>Normal Result (r) when Rounding Mode Is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>To Nearest</td>
</tr>
<tr>
<td>v &lt; −Nmax, p &lt; −Nmax</td>
<td>Overflow</td>
<td>−∞¹</td>
</tr>
<tr>
<td>v &lt; −Nmax, p = −Nmax</td>
<td>Normal</td>
<td>−Nmax</td>
</tr>
<tr>
<td>−Nmax ≤ v ≤ −Nmin</td>
<td>Normal</td>
<td>p</td>
</tr>
<tr>
<td>−Nmin &lt; v ≤ −Dmin</td>
<td>Tiny</td>
<td>d*</td>
</tr>
<tr>
<td>−Dmin &lt; v &lt; −Dmin/2</td>
<td>Tiny</td>
<td>−Dmin</td>
</tr>
<tr>
<td>−Dmin/2 ≤ v &lt; 0</td>
<td>Tiny</td>
<td>−0</td>
</tr>
<tr>
<td>v = 0</td>
<td>Exact zero difference²</td>
<td>+0</td>
</tr>
<tr>
<td>0 &lt; v ≤ +Dmin/2</td>
<td>Tiny</td>
<td>+0</td>
</tr>
<tr>
<td>+Dmin/2 &lt; v &lt; +Dmin</td>
<td>Tiny</td>
<td>+Dmin</td>
</tr>
<tr>
<td>+Dmin ≤ v &lt; +Nmin</td>
<td>Tiny</td>
<td>d*</td>
</tr>
<tr>
<td>+Nmin ≤ v ≤ +Nmax</td>
<td>Normal</td>
<td>p</td>
</tr>
<tr>
<td>+Nmax &lt; v, p = +Nmax</td>
<td>Normal</td>
<td>+Nmax</td>
</tr>
<tr>
<td>+Nmax &lt; v, +Nmax &lt; p</td>
<td>Overflow</td>
<td>+∞¹</td>
</tr>
</tbody>
</table>

**Explanation:**

- This situation cannot occur.
- The rounded value, in the extreme case, may be Nmin. In this case, the exception conditions are underflow, inexact and incremented.
- The exact result r is considered to have been incremented.
- The exact-zero-difference case applies only to ADD, SUBTRACT, MULTIPLY AND ADD, and MULTIPLY AND SUBTRACT. For all other operations, a zero result is detected by inspection of the source operands without use of the R(v) function.
- The value derived when the exact result v is rounded to the format of the target, including both precision and bounded exponent range. Except as explained in note *, this is a denormalized number.
- The value derived when the exact result v is rounded to the precision of the target, but assuming an unbounded exponent range.
- The exact result before rounding, assuming unbounded precision and an unbounded exponent range. For LOAD ROUNDED, v is the source value a.

**Note:**

- Smallest (in magnitude) representable denormalized number in the target format.
- Largest (in magnitude) representable finite number in the target format.
- Smallest (in magnitude) representable normalized number in the target format.

---

*Figure 19-17 (Part 1 of 2). Action for R(v): Rounding and Range Function*
### Table: Action for R(v): Rounding and Range Function

| Case       | Is r Inexact (r≠v) | Overflow Mask (FPC 0.2) | Underflow Mask (FPC 0.3) | Inexact Mask (FPC 0.4) | Is r Incremented (|r|>|v|) | Is p Inexact (p≠v) | Is p Incremented (|p|>|v|) | Results                                      |
|------------|--------------------|-------------------------|--------------------------|------------------------|-----------------------------|------------------------|-----------------------|---------------------------------------------|
| Overflow   | Yes                | 0                       | –                        | 0                      | –                           | –                      | –                     | T(r), SFo ← 1, SFx ← 1                        |
| Overflow   | Yes                | 0                       | –                        | 1                      | No                          | –                      | –                     | T(r), SFo ← 1, PIDx(08)                       |
| Overflow   | Yes                | 0                       | –                        | 1                      | Yes                         | –                      | –                     | T(r), SFo ← 1, PIDy(0C)                       |
| Overflow   | Yes                | 1                       | –                        | –                      | No                          | No                    | Yes                   | Tw(p+β), PIDo(20)                            |
| Overflow   | Yes                | 1                       | –                        | –                      | Yes                         | No                    | Yes                   | Tw(p+β), PIDox(28)                           |
| Normal     | No                 | –                       | –                        | –                      | –                           | –                      | –                     | T(r)                                        |
| Normal     | Yes                | –                       | –                        | 0                      | –                           | –                      | –                     | T(r), SFx ← 1                                |
| Normal     | Yes                | –                       | –                        | 1                      | No                          | –                      | –                     | T(r), PIDx(08)                               |
| Normal     | Yes                | –                       | –                        | 1                      | Yes                         | –                      | –                     | T(r), PIDy(0C)                               |
| Tiny       | No                 | –                       | 0                        | –                      | –                           | –                      | –                     | T(r)                                        |
| Tiny       | Yes                | –                       | 1                        | –                      | No                          | No                    | No                    | Tw(p+β), PIDu(10)                            |
| Tiny       | Yes                | –                       | 0                        | 0                      | –                           | –                      | –                     | T(r), SFu ← 1, SFx ← 1                       |
| Tiny       | Yes                | –                       | 0                        | 1                      | No                          | –                      | –                     | T(r), SFu ← 1, PIDx(08)                       |
| Tiny       | Yes                | –                       | 0                        | 1                      | Yes                         | –                      | –                     | T(r), SFu ← 1, PIDy(0C)                       |
| Tiny       | Yes                | –                       | 1                        | –                      | No                          | No                    | No                    | Tw(p+β), PIDu(10)                            |
| Tiny       | Yes                | –                       | 1                        | –                      | Yes                         | Yes                   | Yes                   | Tw(p+β), PIDuy(1C)                           |

**Explanation:**

- The results do not depend on this condition or mask bit.
- This condition is true by virtue of the state of some condition to the left of this column.
- Wrap adjust, which depends on the type of operation and operand format. For all operations except LOAD ROUNDED, the wrap adjust depends on the target format: \( \beta = 2^\kappa \), where \( \kappa \) is 512 for long and 8192 for extended. For LOAD ROUNDED, the wrap adjust depends on the source format: \( \beta = 2^\alpha \), where \( \alpha \) is 192 for short, 1536 for long, and 24576 for extended.
- The value derived when the exact result \( v \) is rounded to the precision of the target, but assuming an unbounded exponent range.
- Normal result as defined in Part 1 of this figure.
- Exact result before rounding, assuming unbounded precision and unbounded exponent range.
- Program interruption for data exception, condition c, with DXC of h in hex. See [Figure 19-13 on page 19-14](#).
- IEEE overflow flag, FPC 1.2.
- IEEE underflow flag, FPC 1.3.
- IEEE inexact flag, FPC 1.4.
- The value \( x \) is placed at the target operand location.
- The wrapped result \( x \) is placed at the target operand location. For all operations except LOAD ROUNDED, the wrapped result is in the same format and length as normal results at the target location. For LOAD ROUNDED, the wrapped result is in the same format and length as the source, but rounded to the precision of the target.

---

*Figure 19-17 (Part 2 of 2). Action for R(v): Rounding and Range Function*
COMPARE

Mnemonic1 R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>//////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

Mnemonic1 Op Code Operands
CEBR 'B309' Short BFP
CDBR 'B319' Long BFP
CXBR 'B349' Extended BFP

Mnemonic2 R1,Dz(X2,B2) [RXE]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Mnemonic2 Op Code Operands
CEB 'ED09' Short BFP
CDB 'ED19' Long BFP

The first operand is compared with the second operand, and the condition code is set to indicate the result.

If both operands are numeric and finite, the comparison is algebraic and follows the procedure for BFP subtraction, except that the difference is discarded after setting the condition code, and both operands remain unchanged. If the difference is exactly zero with either sign, the operands are equal; this includes zero operands (so +0 equals −0). If a nonzero difference is positive or negative, the first operand is high or low, respectively.

+∞ compares greater than any finite number, and all finite numbers compare greater than −∞. Two infinity operands of like sign compare equal.

Numeric comparison is exact, and the condition code is determined for finite operands as if range and precision were unlimited. No overflow or underflow condition can occur.

If either or both operands are QNaNs and neither operand is an SNaN, the comparison result is unordered, and condition code 3 is set.

If either or both operands are SNaNs, an IEEE-invalid-operation condition is recognized. If the IEEE invalid-operation mask bit is one, a program interruption for a data exception with DXC 80 hex (IEEE invalid operation) occurs. If the IEEE-invalid-operation mask bit is zero, the IEEE-invalid-operation flag bit is set to one, and instruction execution is completed by setting condition code 3.

See Figure 19-18 on page 19-24 for a detailed description of the results of this instruction.

For CXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

Resulting Condition Code:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Operands equal</td>
<td>1</td>
<td>First operand low</td>
<td>2</td>
</tr>
</tbody>
</table>

IEEE Exception Conditions:

- Invalid operation

Program Exceptions:

- Access (fetch, operand 2 of CEB and CDB only)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (CXBR only)

Programming Notes:

1. COMPARE may be used by a compiler to implement those comparisons which are required by the IEEE standard to not recognize an exception condition when the result is unordered due to a QNaN.

2. The IEEE standard requires that it be possible to compare BFP operands in different formats. To accomplish this, LOAD LENGTHENED may be used before COMPARE to convert the shorter operand to the same format as the longer.
Figure 19-18. Results: COMPARE

<table>
<thead>
<tr>
<th>First Operand (a) Is</th>
<th>−∞</th>
<th>−Fc</th>
<th>−0</th>
<th>+0</th>
<th>+Fc</th>
<th>+∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>cc0</td>
<td>cc1</td>
<td>cc1</td>
<td>cc1</td>
<td>cc1</td>
<td>cc1</td>
<td>cc3</td>
<td>Xi: cc3</td>
</tr>
<tr>
<td>−Fc</td>
<td>cc2</td>
<td>C(a:b)</td>
<td>cc1</td>
<td>cc1</td>
<td>cc1</td>
<td>cc1</td>
<td>cc3</td>
<td>Xi: cc3</td>
</tr>
<tr>
<td>−0</td>
<td>cc2</td>
<td>cc2</td>
<td>cc0</td>
<td>cc0</td>
<td>cc1</td>
<td>cc1</td>
<td>cc3</td>
<td>Xi: cc3</td>
</tr>
<tr>
<td>+0</td>
<td>cc2</td>
<td>cc2</td>
<td>cc0</td>
<td>cc0</td>
<td>cc1</td>
<td>cc1</td>
<td>cc3</td>
<td>Xi: cc3</td>
</tr>
<tr>
<td>+Fc</td>
<td>cc2</td>
<td>cc2</td>
<td>cc2</td>
<td>cc2</td>
<td>C(a:b)</td>
<td>cc1</td>
<td>cc3</td>
<td>Xi: cc3</td>
</tr>
<tr>
<td>+∞</td>
<td>cc2</td>
<td>cc2</td>
<td>cc2</td>
<td>cc2</td>
<td>cc2</td>
<td>cc2</td>
<td>cc3</td>
<td>Xi: cc3</td>
</tr>
<tr>
<td>QNaN</td>
<td>cc3</td>
<td>cc3</td>
<td>cc3</td>
<td>cc3</td>
<td>cc3</td>
<td>cc3</td>
<td>cc3</td>
<td>Xi: cc3</td>
</tr>
<tr>
<td>SNaN</td>
<td>Xi: cc3</td>
<td>Xi: cc3</td>
<td>Xi: cc3</td>
<td>Xi: cc3</td>
<td>Xi: cc3</td>
<td>Xi: cc3</td>
<td>Xi: cc3</td>
<td>Xi: cc3</td>
</tr>
</tbody>
</table>

Explanation:
ccn       Condition code is set to n.
C(a:b)    Basic compare results. See Figure 19-19.
Fc        Finite nonzero number (includes both denormalized and normalized).
Xi:       IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.

Figure 19-19. Basic Compare Results

COMPARE AND SIGNAL

Mnemonic1  R1,R2     [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>0</th>
<th>16</th>
<th>24</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
</table>

Mnemonic1  Op Code  Operands
KEBR      'B308'     Short BFP
KDBR      'B318'     Long BFP
KXBR      'B348'     Extended BFP

Relation of Value (a) to Value (b)  Condition Code for C(a:b)

| a=b   | cc0 |
| a<b   | cc1 |
| a>b   | cc2 |

Explanation:
ccn       Condition code is set to n.

Figure 19-19. Basic Compare Results

The first operand is compared with the second operand, and the condition code is set to indicate the result. The operation is the same as for COMPARE except that QNaN operands cause an IEEE-invalid-operation condition to be recognized. Thus, QNaN operands are treated as if they were SNaNs.

See Figure 19-20 on page 19-25 for a detailed description of the results of this instruction.

For KXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

Resulting Condition Code:

| 0 | Operands equal  |
| 1 | First operand low  |
| 2 | First operand high  |
| 3 | Operands unordered  |
IEEE Exception Conditions:
- Invalid operation

Program Exceptions:
- Access (fetch, operand 2 of KEB and KDB only)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (KXBR only)

Programming Notes:
1. COMPARE AND SIGNAL may be used by a compiler to implement those comparisons which are required by the IEEE standard to recognize an exception condition when the result is unordered due to a QNaN.
2. The IEEE standard requires that it be possible to compare BFP operands in different formats. To accomplish this, LOAD LENGTHENED may be used before COMPARE AND SIGNAL to convert the shorter operand to the same format as the longer.

<table>
<thead>
<tr>
<th>First Operand (a) Is</th>
<th>Results for COMPARE AND SIGNAL (a:b) when Second Operand (b) Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>−∞</td>
</tr>
<tr>
<td>−∞</td>
<td>cc0</td>
</tr>
<tr>
<td>−Fm</td>
<td>cc2</td>
</tr>
<tr>
<td>−0</td>
<td>cc2</td>
</tr>
<tr>
<td>+0</td>
<td>cc2</td>
</tr>
<tr>
<td>+Fm</td>
<td>cc2</td>
</tr>
<tr>
<td>+∞</td>
<td>cc2</td>
</tr>
<tr>
<td>NaN</td>
<td>Xi: cc3</td>
</tr>
</tbody>
</table>

Explanation:
- ccn: Condition code is set to n.
- C(a:b): Basic compare results. See Figure 19-19 on page 19-24
- Fn: Finite nonzero number (includes both denormalized and normalized).
- Xi: IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.

Figure 19-20. Results: COMPARE AND SIGNAL
CONVERT FROM FIXED

Mnemonic R₁,R₂  [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEFBR</td>
<td>'B394'</td>
<td>32-bit binary-integer operand, short BFP result</td>
</tr>
<tr>
<td>CDFBR</td>
<td>'B395'</td>
<td>32-bit binary-integer operand, long BFP result</td>
</tr>
<tr>
<td>CXFBR</td>
<td>'B396'</td>
<td>32-bit binary-integer operand, extended BFP result</td>
</tr>
<tr>
<td>CEGBR</td>
<td>'B3A4'</td>
<td>64-bit binary-integer operand, short BFP result</td>
</tr>
<tr>
<td>CDGBR</td>
<td>'B3A5'</td>
<td>64-bit binary-integer operand, long BFP result</td>
</tr>
<tr>
<td>CXGBR</td>
<td>'B3A6'</td>
<td>64-bit binary-integer operand, extended BFP result</td>
</tr>
</tbody>
</table>

The fixed-point second operand is converted to the BFP format, and the result is placed at the first-operand location.

The second operand is a signed binary integer that is located in the general register designated by R/basesupertwo. A 32-bit operand is in bit positions 32-63 of the register.

The result is rounded according to the current rounding mode before it is placed at the first-operand location.

See Figure 19-21 on page 19-27 for a detailed description of the results of this instruction.

For CXFBR and CXGBR, the R₁ field must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:**
- Inexact (CEFBR, CDGBR, CEGBR)

**Program Exceptions:**
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (CXFBR and CXGBR)

---

CONVERT TO FIXED

Mnemonic R₁,M₃,R₂  [RRF]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>M₃</th>
<th>////////</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEFBR</td>
<td>'B398'</td>
<td>Short BFP operand, 32-bit binary-integer result</td>
</tr>
<tr>
<td>CFDBR</td>
<td>'B399'</td>
<td>Long BFP operand, 32-bit binary-integer result</td>
</tr>
<tr>
<td>CFXBR</td>
<td>'B3A8'</td>
<td>Extended BFP operand, 32-bit binary-integer result</td>
</tr>
<tr>
<td>CGEBR</td>
<td>'B3AA'</td>
<td>Short BFP operand, 64-bit binary-integer result</td>
</tr>
<tr>
<td>CGDBR</td>
<td>'B3A9'</td>
<td>Long BFP operand, 64-bit binary-integer result</td>
</tr>
<tr>
<td>CGXBR</td>
<td>'B3AB'</td>
<td>Extended BFP operand, 64-bit binary-integer result</td>
</tr>
</tbody>
</table>

The BFP second operand is rounded to an integer value and then converted to the fixed-point format. The result is placed at the first-operand location.

The result is a signed binary integer that is placed in the general register designated by R₁. A 32-bit result replaces bits 32-63 of the register, and bits 0-31 of the register remain unchanged.

If the second operand is numeric and finite, it is rounded to an integer value by rounding as specified by the modifier in the M₃ field:

**M₃ Rounding Method**
- 0 According to current rounding mode
- 1 Biased round to nearest
- 4 Round to nearest
- 5 Round toward 0
- 6 Round toward +∞
- 7 Round toward −∞

A modifier other than 0, 1, or 4-7 is invalid.

When the modifier field is zero, rounding is controlled by the current rounding mode specified in the FPC register. When the field is not zero, rounding is performed as specified by the modifier, regardless of the current rounding mode. Rounding for modifiers 4-7 is the same as for rounding modes 0-3 (binary 00-11), respectively. Biased round to nearest (modifier 1) is the same as round to nearest (modifier 4), except when the second operand is exactly halfway between two integers, in which case the result for biased rounding is the next integer that is greater in magnitude.
Table 19.1: Results for Instructions with a Single Operand (a)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result for −∞</th>
<th>−Fn</th>
<th>−0</th>
<th>+0</th>
<th>+Fn</th>
<th>+∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVERT FROM FIXED</td>
<td>−</td>
<td>Rf(a)</td>
<td>−</td>
<td>T(+0)</td>
<td>Rf(a)</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>LOAD AND TEST</td>
<td>T(−∞)</td>
<td>T(a)</td>
<td>T(−0)</td>
<td>T(+0)</td>
<td>T(a)</td>
<td>T(+)</td>
<td>T(a)</td>
<td>Xi: T(a*)</td>
</tr>
<tr>
<td>LOAD LENGTHENED</td>
<td>T(−∞)</td>
<td>T(a)</td>
<td>T(−0)</td>
<td>T(+0)</td>
<td>T(a)</td>
<td>T(+)</td>
<td>T(a)</td>
<td>Xi: T(a*)</td>
</tr>
<tr>
<td>LOAD ROUNDED</td>
<td>T(−∞)</td>
<td>R(a)</td>
<td>T(−0)</td>
<td>T(+0)</td>
<td>R(a)</td>
<td>T(+)</td>
<td>T(a)</td>
<td>Xi: T(a*)</td>
</tr>
<tr>
<td>SQUARE ROOT</td>
<td>Xi: T(dNaN)</td>
<td>Xi: T(dNaN)</td>
<td>T(−0)</td>
<td>T(+0)</td>
<td>R(√ a)</td>
<td>T(+)</td>
<td>T(a)</td>
<td>Xi: T(a*)</td>
</tr>
</tbody>
</table>

**Explanation:**

- This situation cannot occur.
- The SNaN is converted to the corresponding QNaN before it is placed at the target operand location.
- The operand is extended to the longer format by appending zeros on the right before it is placed at the target operand location.
- The NaN is shortened to the target format by truncating the rightmost bits.
- dNaN: Default quiet NaN.
- Fn: Finite nonzero number (includes both denormalized and normalized).
- R(v): Rounding and range action is performed on the value v. See Figure 19-17 on page 19-21.
- Rf(a): The value a is converted to the exact floating-point number v, and then action R(v) is performed.
- T(x): The value x is placed in the target operand location.
- Xi: IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.

**Figure 19-21. Results: Single-Operand Instructions**

The sign of the result is the sign of the second operand, except that a zero result has a plus sign.

See Figure 19-22 on page 19-28 for a detailed description of the results of this instruction.

The M₃ field must designate a valid modifier; otherwise, a specification exception is recognized.

For CFXBR and CGXBR, the R₂ field must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0  Source was zero
1  Source was less than zero
2  Source was greater than zero
3  Special case

**IEEE Exception Conditions:**

- Invalid operation
- Inexact

**Program Exceptions:**

- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification
<table>
<thead>
<tr>
<th>Operand (a)</th>
<th>Is n Inexact (n=a)</th>
<th>Inv.-Op. Mask (FPC 0.0)</th>
<th>Inexact Mask (FPC 0.4)</th>
<th>Is n Incremented (In&gt;ial)</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞ ≤ a &lt; MN, p &lt; MN</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>T(MN), SFi←1, SFx←1, cc3</td>
</tr>
<tr>
<td>−∞ ≤ a &lt; MN, p &lt; MN</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>–</td>
<td>T(MN), SFi←1, cc3, PIDx(08)</td>
</tr>
<tr>
<td>−∞ ≤ a &lt; MN, p = MN</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>PIDi(80)</td>
</tr>
<tr>
<td>−∞ a &lt; MN, p = MN</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>–</td>
<td>T(MN), SFx←1, cc1</td>
</tr>
<tr>
<td>MN ≤ a &lt; 0</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>T(n), cc1</td>
</tr>
<tr>
<td>MN ≤ a &lt; 0</td>
<td>Yes</td>
<td>–</td>
<td>0</td>
<td>–</td>
<td>T(n), SFx←1, cc1</td>
</tr>
<tr>
<td>MN ≤ a &lt; 0</td>
<td>Yes</td>
<td>–</td>
<td>1</td>
<td>No</td>
<td>T(n), cc1, PIDx(08)</td>
</tr>
<tr>
<td>MN ≤ a &lt; 0</td>
<td>Yes</td>
<td>–</td>
<td>1</td>
<td>Yes</td>
<td>T(n), cc1, PIDy(0C)</td>
</tr>
<tr>
<td>−0</td>
<td>No^1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>T(0), cc0</td>
</tr>
<tr>
<td>+0</td>
<td>No^1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>T(0), cc0</td>
</tr>
<tr>
<td>0 ≤ a ≤ MP</td>
<td>No</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>T(n), cc2</td>
</tr>
<tr>
<td>0 ≤ a ≤ MP</td>
<td>Yes</td>
<td>–</td>
<td>0</td>
<td>–</td>
<td>T(n), SFx←1, cc2</td>
</tr>
<tr>
<td>0 ≤ a ≤ MP</td>
<td>Yes</td>
<td>–</td>
<td>1</td>
<td>No</td>
<td>T(n), cc2, PIDx(08)</td>
</tr>
<tr>
<td>0 ≤ a ≤ MP</td>
<td>Yes</td>
<td>–</td>
<td>1</td>
<td>Yes</td>
<td>T(n), cc2, PIDy(0C)</td>
</tr>
<tr>
<td>MP ≤ a &lt; +∞, p = MP</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>–</td>
<td>T(MP), SFx←1, cc2</td>
</tr>
<tr>
<td>MP ≤ a &lt; +∞, p = MP</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>T(MP), cc2, PIDx(08)</td>
</tr>
<tr>
<td>MP ≤ a &lt; +∞, p &gt; MP</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>T(MP), SFi←1, SFx←1, cc3</td>
</tr>
<tr>
<td>MP ≤ a &lt; +∞, p &gt; MP</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>–</td>
<td>T(MP), SFi←1, cc3, PIDx(08)</td>
</tr>
<tr>
<td>MP ≤ a &lt; +∞, p &gt; MP</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>PIDi(80)</td>
</tr>
<tr>
<td>NaN</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td>T(MN), SFi←1, SFx←1, cc3</td>
</tr>
<tr>
<td>NaN</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>–</td>
<td>T(MN), SFi←1, cc3, PIDx(08)</td>
</tr>
<tr>
<td>NaN</td>
<td>–</td>
<td>1</td>
<td>–</td>
<td>–</td>
<td>PIDi(80)</td>
</tr>
</tbody>
</table>

**Explanation:**

– The results do not depend on this condition or mask bit.

1 This condition is true by virtue of the state of some condition to the left of this column.

ccn Condition code is set to n.

n The value p converted to a fixed-point result.

p The value derived when the source value a is rounded to an integer using the specified rounding mode.

MN Maximum negative number representable in the target fixed-point format.

MP Maximum positive number representable in the target fixed-point format.

PIDc(h) Program interruption for data exception, condition c, with DXC of h in hex. See Figure 19-13 on page 19-14.

SFi IEEE invalid-operation flag, FPC 1.0.

SFx IEEE inexact flag, FPC 1.4.

T(x) The value x is placed at the target operand location.

---

**Figure 19-22. Results: CONVERT TO FIXED**
DIVIDE

Mnemonic1 R1, R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBR</td>
<td>'B30D'</td>
<td>Short BFP</td>
</tr>
<tr>
<td>DDBR</td>
<td>'B31D'</td>
<td>Long BFP</td>
</tr>
<tr>
<td>DXBR</td>
<td>'B34D'</td>
<td>Extended BFP</td>
</tr>
</tbody>
</table>

Mnemonic2 R1, D2 (X2, B2) [RXE]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic2</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEB</td>
<td>'ED0D'</td>
<td>Short BFP</td>
</tr>
<tr>
<td>DDB</td>
<td>'ED1D'</td>
<td>Long BFP</td>
</tr>
</tbody>
</table>

The first operand (the dividend) is divided by the second operand (the divisor), and the quotient is placed at the first-operand location. No remainder is preserved.

If the divisor is nonzero and both the dividend and divisor are numeric and finite, the first operand is divided by the second operand to form an intermediate quotient. The intermediate quotient, if nonzero, is normalized and rounded to the target format according to the current rounding mode.

The sign of the quotient is the exclusive or of the operand signs. This includes the sign of a zero quotient.

If the divisor is zero but the dividend is nonzero and finite, an IEEE-division-by-zero condition is recognized. If the dividend and divisor are both zero, or if both are infinite, regardless of sign, an IEEE-invalid-operation condition is recognized.

See [Figure 19-23 on page 19-30] for a detailed description of the results of this instruction.

For DXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:**
- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact

**Program Exceptions:**
- Access (fetch, operand 2 of DEB and DDB only)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (DXBR only)

DIVIDE TO INTEGER

Mnemonic R1, R3, R2, M4 [RRF]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R3</th>
<th>M4</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIEBR</td>
<td>'B353'</td>
<td>Short BFP</td>
</tr>
<tr>
<td>DIDBR</td>
<td>'B35B'</td>
<td>Long BFP</td>
</tr>
</tbody>
</table>

The first operand (the dividend) is divided by the second operand (the divisor). An integer quotient in BFP form is produced and placed at the third-operand location. The remainder replaces the dividend at the first-operand location. The first, second, and third operands must be in different registers. The condition code indicates whether partial or complete results have been produced and whether the quotient is numeric and finite.

The remainder result is
\[ r = a - b \cdot n \]

where \( a \) is the dividend, \( b \) the divisor, and \( n \) an integer obtained by rounding the precise quotient \( q = a/b \).

The first-operand result is \( r \) with the sign determined by the above expression. The third-operand result is \( n \) with a sign that is the exclusive or of the dividend and divisor signs.

If the precise quotient is not an integer and the two integers closest to this precise quotient cannot both be represented exactly in the precision of the quotient, then a partial quotient and partial remainder are formed. This partial quotient \( n \) and the corresponding partial remainder...
<table>
<thead>
<tr>
<th>Dividend (a)</th>
<th>$-\infty$</th>
<th>$-\text{Fn}$</th>
<th>$-0$</th>
<th>$+0$</th>
<th>$+\text{Fn}$</th>
<th>$+\infty$</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>Xi: T(dNaN)</td>
<td>T($+\infty$)</td>
<td>T($+\infty$)</td>
<td>T($-\infty$)</td>
<td>T($-\infty$)</td>
<td>Xi: T(dNaN)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>$-\text{Fn}$</td>
<td>T($+0$)</td>
<td>R(a$\div$b)</td>
<td>XZ: T($+\infty$)</td>
<td>XZ: T($-\infty$)</td>
<td>R(a$\div$b)</td>
<td>T($-0$)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>$-0$</td>
<td>T($+0$)</td>
<td>T($+0$)</td>
<td>Xi: T(dNaN)</td>
<td>Xi: T(dNaN)</td>
<td>T($-0$)</td>
<td>T($-0$)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>$+0$</td>
<td>T($-0$)</td>
<td>T($-0$)</td>
<td>Xi: T(dNaN)</td>
<td>Xi: T(dNaN)</td>
<td>T($+0$)</td>
<td>T($+0$)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>$+\text{Fn}$</td>
<td>T($-0$)</td>
<td>R(a$\div$b)</td>
<td>XZ: T($-\infty$)</td>
<td>XZ: T($+\infty$)</td>
<td>R(a$\div$b)</td>
<td>T($+0$)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>Xi: T(dNaN)</td>
<td>T($-\infty$)</td>
<td>T($-\infty$)</td>
<td>T($+\infty$)</td>
<td>T($+\infty$)</td>
<td>Xi: T(dNaN)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>SNaN</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
</tr>
</tbody>
</table>

**Explanation:**

- The SNaN is converted to the corresponding QNaN before it is placed at the target operand location.
- **Fn** Finite nonzero number (includes both denormalized and normalized).
- **R(v)** Rounding and range action is performed on the value v. See Figure 19-17 on page 19-21.
- **T(x)** The value x is placed at the target operand location.
- **Xi:** IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.
- **XZ:** IEEE division-by-zero exception. The results shown are produced only when FPC 0.1 is zero.

Figure 19-23. Results: DIVIDE

\[ r = a - b \cdot n \]

are used as the results. The sign of a partial remainder is the same as the sign of the dividend. The sign of a partial quotient is the exclusive or of the dividend and divisor signs.

If the remainder is zero, then the precise quotient is an integer and can be represented exactly in the precision of the quotient.

The M₄ field, called the modifier field, specifies rounding of the final quotient. This rounding is called the “specified quotient rounding mode” as contrasted to the “current rounding mode” specified by the rounding-mode bits in the FPC register. The final quotient is rounded according to the specified quotient rounding mode. The specified quotient rounding mode affects only the final quotient; partial quotients are rounded toward zero.

Since the partial quotient is rounded toward zero, the partial remainder is always exact. For the specified quotient rounding modes of round toward 0, round to nearest, and biased round to nearest, the final remainder is exact. For the specified quotient rounding modes of round toward $+\infty$ and round toward $-\infty$, the final remainder may not be exact.

The final quotient is rounded to an integer by rounding as specified by the modifier in the M₄ field:

**M₄ Rounding Method**

- 0 According to current rounding mode
- 1 Biased round to nearest
- 4 Round to nearest
- 5 Round toward 0
- 6 Round toward $+\infty$
- 7 Round toward $-\infty$
A modifier other than 0, 1, or 4-7 is invalid.

When the modifier field is zero, rounding of the final quotient is controlled by the current rounding mode specified in the FPC register. When the field is not zero, rounding is performed as specified by the modifier, regardless of the current rounding mode. Rounding for modifiers 4-7 is the same as for rounding modes 0-3 (binary 00-11), respectively. Biased round to nearest (modifier 1) is the same as round to nearest (modifier 4), except when the final quotient is exactly halfway between two integers, in which case the result for biased rounding is the next integer that is greater in magnitude.

Underflow is recognized only on the final remainder, not on the partial remainder.

For the specified quotient rounding modes of round toward $+\infty$ and round toward $-\infty$, the final remainder may not be exact. When, in these cases, the final remainder is inexact, it is rounded according to the current rounding mode specified in the FPC register.

The sign of a zero quotient is the exclusive or of the divisor and dividend signs.

A zero remainder has the sign of the dividend.

See Figure 19-24 on page 19-32 for a detailed description of the results of this instruction.

If the quotient exponent is greater than the largest exponent that can be represented in the operand format, the correct remainder or partial remainder still is produced, and the third-operand result is the correct value, but with the exponent reduced by 192 or 1536 for short or long operands, respectively. The condition code indicates this out-of-range condition.

The $M_4$ field must designate a valid modifier, and the $R_1$, $R_2$, and $R_3$ fields must designate different registers; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0 Remainder complete; normal quotient
1 Remainder complete; quotient overflow or NaN
2 Remainder incomplete; normal quotient
3 Remainder incomplete; quotient overflow or NaN

**IEEE Exception Conditions:**

- Invalid operation
- Underflow
- Inexact

**Program Exceptions:**

- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification

**Programming Notes:**

1. The Remainder operation, as defined in the IEEE standard, is produced by issuing DIVIDE TO INTEGER in an iterative loop, with the $M_4$ field set to 4.

2. The rounding specifications of round to nearest, round toward 0, and round toward $-\infty$ permit the instruction to be used directly to produce the Remainder, MOD, and modulo functions, respectively.

3. When DIVIDE TO INTEGER is used in an iterative loop, all quotients are produced in BFP format but may be considered as portions of a multiple-precision fixed-point number.

4. In the case when the resulting remainder is denormalized, the IEEE standard requires that if traps are implemented and the underflow mask is one, then an underflow trap must occur. To accomplish this, DIVIDE TO INTEGER recognizes underflow on the final remainder but not on the partial remainder. Since in all cases when underflow occurs on the partial remainder it will occur again on the final remainder, recognizing overflow on only the final remainder avoids two underflow traps to be reported for what the standard considers a single Remainder operation.
<table>
<thead>
<tr>
<th>Dividend (a)</th>
<th>$-\infty$</th>
<th>$-\text{Fn}$</th>
<th>$-0$</th>
<th>$+0$</th>
<th>$+\text{Fn}$</th>
<th>$+\infty$</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>T(b), cc1</td>
<td>Xi: T(b*), cc1</td>
<td></td>
</tr>
<tr>
<td>$-\text{Fn}$</td>
<td>T(a,+0), cc0</td>
<td>D(a,b)</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>D(a,b)</td>
<td>T(a,−0), cc0</td>
<td>T(b), cc1</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>T(−0,+0), cc0</td>
<td>T(−0,+0), cc0</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>T(−0,−0), cc0</td>
<td>T(−0,−0), cc0</td>
<td>T(b), cc1</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>T(+0,−0), cc0</td>
<td>T(+0,−0), cc0</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>T(+0,−0), cc0</td>
<td>T(+0,−0), cc0</td>
<td>T(b), cc1</td>
<td></td>
</tr>
<tr>
<td>$+\text{Fn}$</td>
<td>T(a,−0), cc0</td>
<td>D(a,b)</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>D(a,b)</td>
<td>T(a,+0), cc0</td>
<td>T(b), cc1</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>Xi: T(dNaN), cc1</td>
<td>T(b), cc1</td>
<td>Xi: T(b*), cc1</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>T(a), cc1</td>
<td>T(a), cc1</td>
<td>T(a), cc1</td>
<td>T(a), cc1</td>
<td>T(a), cc1</td>
<td>T(a), cc1</td>
<td>T(a), cc1</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>Xi: T(a*), cc1</td>
<td>Xi: T(a*), cc1</td>
<td>Xi: T(a*), cc1</td>
<td>Xi: T(a*), cc1</td>
<td>Xi: T(a*), cc1</td>
<td>Xi: T(a*), cc1</td>
<td>Xi: T(a*), cc1</td>
<td></td>
</tr>
</tbody>
</table>

**Explanation:**

- The SNaN is converted to the corresponding QNaN before it is placed at the target operand location.
- Condition code is set to n.
- D(a,b) Basic divide-to-integer results. See Part 2 of this figure.
- Fn Finite nonzero number (includes both denormalized and normalized).
- T(r,q) Results r (the remainder) and q (the quotient) are placed in target operands 1 and 3, respectively.
- T(x) Value x is placed in both target operands 1 and 3.
- Xi: IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.

*Figure 19-24 (Part 1 of 2). Results: DIVIDE TO INTEGER*
### Explanation:

1. The results do not depend on this condition or mask bit.
2. This condition is true by virtue of the state of some condition to the left of this column. That is, when \(|q| < 2^P\), there cannot be a quotient overflow; the cases of remainder is zero, tiny, or inexact are mutually exclusive; and when \(r\) is exact, it is not incremented.
3. Underflow is not recognized for a partial remainder.
4. Wrap adjust, which depends on the target format: \(\beta = 2^\alpha\), where \(\alpha\) is 192 for short and 1536 for long.
5. The absolute value of \(q\), where \(q\) is the exact result of \(a \div b\) before rounding, assuming unbounded precision and unbounded exponent range.
6. Condition code is set to 0 (remainder complete; normal quotient).
7. Condition code is set to 1 (remainder complete; quotient overflow).
8. Condition code is set to 2 (remainder incomplete; normal quotient).
9. Condition code is set to 3 (remainder incomplete; quotient overflow).
10. Integer quotient. \(n = q\), rounded toward 0 for partial results and rounded according to the specified quotient rounding mode for final results. The sign of the integer quotient, including the cases of partial and final, wrapped-around overflow and zero, is the exclusive or of the signs of the dividend (a) and divisor (b).
11. Remainder. \(r = a - b \cdot n\). A partial remainder is always exact; no rounding is necessary. The sign of a partial remainder is always the same as the sign of the dividend (a). A final remainder is rounded according to the current rounding mode (if necessary). The sign of a zero remainder is the same as the sign of the dividend (a). The sign of a nonzero final remainder is determined by the rules of algebra.
12. Precision of the operand, which depends on the target format: \(P = 24\) for short and \(53\) for long.
13. Program interruption for data exception, condition c, with DXC of h in hex. See Figure 19-13 on page 19-14.

### EXTRACT FPC

<table>
<thead>
<tr>
<th>EFPC</th>
<th>R1</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B38C'</td>
<td>// // // // //</td>
<td>R1 // //</td>
</tr>
</tbody>
</table>
0 16 24 28 31

The contents of the FPC (floating-point-control) register are placed in bit positions 32-63 of the general register designated by \(R_1\). Bit positions 0-31 of the general register remain unchanged.
**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:** None.

**Program Exceptions:**
- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)

### LOAD AND TEST

Mnemonic `R1,R2` [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>//////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
</table>

The second operand is placed at the first-operand location, and its sign and magnitude are tested to determine the setting of the condition code. The condition code is set the same as for a comparison of the second operand with zero.

The second operand is placed unchanged at the first-operand location. If the second operand is an SNaN, an IEEE-invalid-operation condition is recognized; if there is no interruption, the result is the corresponding QNaN.

See [Figure 19-21 on page 19-27](#) for a detailed description of the results of this instruction.

For LTXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

#### Resulting Condition Code:

0 Result is zero  
1 Result is less than zero  
2 Result is greater than zero  
3 Result is a NaN

**IEEE Exception Conditions:** Invalid operation

### LOAD COMPLEMENT

Mnemonic `R1,R2` [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>//////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
</table>

The second operand is placed at the first-operand location with the sign bit inverted.

The sign bit is inverted even if the operand is zero. The rest of the second operand is placed unchanged at the first-operand location. The sign is inverted for any operand, including a QNaN or SNaN, without causing an arithmetic exception.

For LCXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

#### Resulting Condition Code:

0 Result is zero  
1 Result is less than zero  
2 Result is greater than zero  
3 Result is a NaN

**IEEE Exception Conditions:** None.

**Program Exceptions:**
- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)
- Specification (LCXBR only)

**Programming Note:** The IEEE standard makes it optional whether operations such as LOAD AND TEST signal invalid operation when the operand is an SNaN. TEST DATA CLASS may be used to test an operand if signaling is not desired.
used in conjunction with this instruction if signalling is desired.

LOAD FP INTEGER

Mnemonic  Rₙ,Mₙ,R₂  [BFP]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>Mₙ</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>31</td>
</tr>
</tbody>
</table>

Mnemonic  Op Code  Operands
FIEBR  'B357'  Short BFP
FIDBR  'B35F'  Long BFP
FIXBR  'B347'  Extended BFP

The second operand is rounded to an integer value in the same floating-point format, and the result is placed at the first-operand location.

The second operand, if numeric, is rounded to an integer value as specified by the modifier in the Mₙ field:

Mₙ  Rounding Method
0  According to current rounding mode
1  Biased round to nearest
4  Round to nearest
5  Round toward 0
6  Round toward +∞
7  Round toward −∞

A modifier other than 0, 1, or 4-7 is invalid.

When the modifier field is zero, rounding is controlled by the current rounding mode in the FPC register. When the field is not zero, rounding is performed as specified by the modifier, regardless of the current rounding mode. Rounding for modifiers 4-7 is the same as for rounding modes 0-3 (binary 00-11), respectively. Biased round to nearest (modifier 1) is the same as round to nearest (modifier 4), except when the second operand is exactly halfway between two integers, in which case the result for biased rounding is the next integer that is greater in magnitude.

In the absence of an interruption, if the second operand is an infinity or a QNaN, the result is that operand; if the second operand is an SNaN, the result is the corresponding QNaN.

The sign of the result is the sign of the second operand, even when the result is zero.

See Figure 19-25 on page 19-36 for a detailed description of the results of this instruction.

The Mₙ field must designate a valid modifier, and, for FIXBR, the R fields must designate valid floating-point-register pairs. Otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

IEEE Exception Conditions:
- Invalid operation
- Inexact

Program Exceptions:
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification

Programming Notes:
1. LOAD FP INTEGER rounds a BFP number to an integer value. These integers, which remain in the BFP format, should not be confused with binary integers, which have a fixed-point format.
2. If the BFP operand is numeric with a large enough exponent so that it is already an integer, the result value remains the same.
### Table: Results: LOAD FP INTEGER

| Operand (a) | Is n Inexact (n≠a) | Inv.-Op. Mask (FPC 0.0) | Inexact Mask (FPC 0.4) | Is n Incremented (ln|lal) | Results |
|-------------|-------------------|------------------------|------------------------|--------------------------|---------|
| −∞         | No¹               | −                      | −                      | −                        | T(−∞)   |
| −Fn        | No                | −                      | −                      | −                        | T(n)    |
| −Fn        | Yes               | −                      | 0                     | −                        | T(n), SFx←1 |
| −Fn        | Yes               | −                      | 1                     | No                       | T(n), PIDx(08) |
| −0         | No¹               | −                      | −                      | −                        | T(0)    |
| +0         | No¹               | −                      | −                      | −                        | T(+0)   |
| +Fn        | No                | −                      | −                      | −                        | T(n)    |
| +Fn        | Yes               | −                      | 0                     | −                        | T(n), SFx←1 |
| +Fn        | Yes               | −                      | 1                     | No                       | T(n), PIDx(08) |
| +∞         | No¹               | −                      | −                      | −                        | T(+∞)   |
| QNaN       | No¹               | −                      | −                      | −                        | T(a)    |
| SNaN       | No¹               | 0                     | −                      | −                        | T(a*), SFi←1 |
| SNaN       | No¹               | 1                     | −                      | −                        | PIDi(80) |

**Explanation:**

− The results do not depend on this condition or mask bit.
* The SNaN is converted to the corresponding QNaN before it is placed at the target operand location.
¹ This condition is true by virtue of the state of some condition to the left of this column.

n The value derived when the source value, a, is rounded to an integer using the specified rounding mode.

Fn Finite nonzero number (includes both denormalized and normalized).
PIDc(h) Program interruption for data exception, condition c, with DXC of h in hex. See [Figure 19-13 on page 19-14](#).

SFx IEEE inexact flag, FPC 1.4.

T(x) The value x is placed at the target operand location.

---

**Figure 19-25. Results: LOAD FP INTEGER**

**LOAD FPC**

<table>
<thead>
<tr>
<th>LFPC</th>
<th>D₂(B₂)</th>
<th>[S]</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B29D'</td>
<td>B₂</td>
<td>D₂</td>
</tr>
</tbody>
</table>

0 16 20 31

The four-byte second operand in storage is loaded into the FPC (floating-point-control) register.

Bits corresponding to unassigned bit positions in the FPC register must be zero; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.
IEEE Exception Conditions: None.

Program Exceptions:
- Access (fetch, operand 2)
- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)
- Specification

LOAD LENGTHENED

Mnemonic R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Mnemonic1 Op Code R/basesuperone R/basesupertwo [RRE]

LDEBR 'B304' Short BFP operand 2, long BFP operand 1
LXDBR 'B305' Long BFP operand 2, extended BFP operand 1
LXEPR 'B306' Short BFP operand 2, extended BFP operand 1

The second operand is extended to a longer format, and the result is placed at the first-operand location.

The sign of the result is the same as the sign of the source. The exponent of the second operand is converted to the corresponding exponent in the result format, and the fraction is extended by appending zeros on the right. If the second operand is an infinity, the result is an infinity of the same sign. If the second operand is an SNaN, an IEEE-invalid-operation condition is recognized; if there is no interruption, the result is the corresponding QNaN with the fraction extended.

For LXDB, LXDBR, LXEPR, and LXEBR, the R fields must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.

IEEE Exception Conditions:
- Invalid operation

Program Exceptions:
- Access (fetch, operand 2 of LDEB, LXEPR, and LXDB only)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (LXEPR, LXEBR, LXDB, LXDBR)

LOAD NEGATIVE

Mnemonic R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>Xz</th>
<th>Bz</th>
<th>Dz</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Mnemonic1 Op Code R/basesuperone X/basesupertwo B/basesupertwo [RXE]

LNEBR 'B301' Short BFP
LNDBR 'B311' Long BFP
LNXBR 'B341' Extended BFP

The second operand is placed at the first-operand location with the sign bit made one.

The sign bit is made one even if the operand is zero. The rest of the second operand is placed unchanged at the first-operand location. The sign is set for any operand, including a QNaN or SNaN, without causing an arithmetic exception.

For LNXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

Resulting Condition Code:
0 Result is zero
1 Result is less than zero
2 --
3 Result is a NaN

IEEE Exception Conditions: None.
Program Exceptions:
- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)
- Specification (LNXBR only)

Programming Note: The IEEE standard makes it optional whether operations such as LOAD NEGATIVE signal invalid operation when the operand is an SNaN. LOAD AND TEST may be used in conjunction with this instruction if signaling is desired.

LOAD POSITIVE

Mnemonic R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>//////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
LPEBR 'B300' Short BFP
LPDBR 'B310' Long BFP
LPXBR 'B340' Extended BFP

The second operand is placed at the first-operand location with the sign bit made zero.

The sign bit is made zero, and the rest of the second operand is placed unchanged at the first-operand location. The sign is set for any operand, including a QNaN or SNaN, without causing an arithmetic exception.

For LPXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

Resulting Condition Code:
0 Result is zero
1 --
2 Result is greater than zero
3 Result is a NaN

IEEE Exception Conditions: None.

Program Exceptions:
- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)
- Specification (LNXBR only)

Programming Note: The IEEE standard makes it optional whether operations such as LOAD POSITIVE signal invalid operation when the operand is an SNaN. LOAD AND TEST may be used in conjunction with this instruction if signaling is desired.

LOAD ROUNDED

Mnemonic R1,R2 [RRE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>//////////</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16 24 28 31</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mnemonic Op Code Operands
LEDBR 'B344' Long BFP source, short BFP target
LDXBR 'B345' Extended BFP source, long BFP target
LEXBR 'B346' Extended BFP source, short BFP target

The second operand, in the format of the source, is rounded to the precision of the target, and the result is placed at the first-operand location. The sign of the result is the same as the sign of the second operand.

The second operand, if numeric, is rounded to the precision of the target fraction according to the current rounding mode. Normally, the result is in the format and length of the target. However, when an IEEE overflow or an IEEE underflow occurs and the corresponding mask bit is one, the operation is completed by producing a wrapped result in the same format and length as the source but rounded to the precision of the target. A short-format result replaces the leftmost 32 bits of the target register, and the rightmost 32 bit positions of the target register remain unchanged. A long-format result is placed in a floating-point register, and the other register of the floating-point register pair, if any, remains unchanged. An extended-format result is placed in a floating-point register pair.

See Figure 19-21 on page 19-27 for a detailed description of the results of this instruction.

For LDXBR and LEXBR, the R1 and R2 fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

Condition Code: The code remains unchanged.
**IEEE Exception Conditions:**
- Invalid operation
- Overflow
- Underflow
- Inexact

**Program Exceptions:**
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (LDXBR and LEXBR)

**Programming Notes:**
1. The sign of the rounded result is the same as the sign of the operand, even when the result is zero.
2. The $R_1$ field for LDXBR and LEXBR must designate a valid floating-point-register pair since in certain cases the result is in the extended format. In normal operation for LDXBR and LEXBR, the result format is long or short, respectively, and this result replaces the leftmost 32 bits or 64 bits of the target-register pair. However, when an IEEE overflow or an IEEE underflow occurs and the corresponding mask bit is one, the operation is completed by placing a result in the extended format at the target location. Thus, the program must take into account the fact that these instructions sometimes update both registers of the pair.

**MULTIPLY**

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>$R_1, R_2$</th>
<th>[RRE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code</td>
<td>//////////</td>
<td>$R_1$</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic1</th>
<th>Op Code</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEEBR</td>
<td>'B317'</td>
<td>Short BFP</td>
</tr>
<tr>
<td>MDBR</td>
<td>'B31C'</td>
<td>Long BFP</td>
</tr>
<tr>
<td>MXBR</td>
<td>'B34C'</td>
<td>Extended BFP</td>
</tr>
<tr>
<td>MDEBR</td>
<td>'B30C'</td>
<td>Short BFP multiplier and multiplicand, long BFP product</td>
</tr>
<tr>
<td>MXDBR</td>
<td>'B307'</td>
<td>Long BFP multiplier and multiplicand, extended BFP product</td>
</tr>
</tbody>
</table>

The product of the second operand (the multiplier) and the first operand (the multiplicand) is placed at the first-operand location.

The two BFP operands, if numeric and finite, are multiplied, forming an intermediate product. For MDEB, MDEBR, MXDB, and MXDBR, the intermediate product is converted to the longer target format; the result cannot overflow or underflow and is exact. For MDB, MDBR, MEEB, MEEBR, and MXBR, the result is rounded to the operand format according to the current rounding mode. For MEEB and MEEBR, the result, as for all short-format results, replaces the leftmost 32 bits of the target register, and the rightmost 32 bit positions of the target register remain unchanged.

The sign of the product, if the product is numeric, is the exclusive or of the operand signs. This includes the sign of a zero or infinite product.

If one operand is a zero and the other an infinity, an IEEE-invalid-operation condition is recognized.

See [Figure 19-26 on page 19-40](#) for a detailed description of the results of this instruction.

The $R_1$ field for MXDB, MXDBR, and MXBR, and the $R_2$ field for MXBR, must designate valid floating-point-register pairs. Otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:**
- Invalid operation
- Overflow (MDB, MDBR, MEEB, MEEBR, MXBR)
- Underflow (MDB, MDBR, MEEB, MEEBR, MXBR)
Inexact (MDB, MDBR, MEEB, MEEBR, MXBR)

Program Exceptions:
- Access (fetch, operand 2 of MDEB, MEEB, MDB, and MXDB only)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)

Programming Note: Interchanging the two operands in a BFP multiplication does not affect the value of the product when the result is numeric. This is not true, however, when both operands are QNaNs, in which case the result is the first operand; or when both operands are SNaNs and the IEEE-invalid-operation mask bit in the FPC register is zero, in which case the result is the QNaN derived from the first operand.

<table>
<thead>
<tr>
<th>First Operand (a) Is</th>
<th>−∞</th>
<th>−Fn</th>
<th>−0</th>
<th>+0</th>
<th>+Fn</th>
<th>+∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>−∞</td>
<td>T(+)</td>
<td>T(+)</td>
<td>Xi: T(dNaN)</td>
<td>Xi: T(dNaN)</td>
<td>T(−∞)</td>
<td>T(−∞)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>−Fn</td>
<td>T(+)</td>
<td>R(a·b)</td>
<td>T(+0)</td>
<td>T(−0)</td>
<td>R(a·b)</td>
<td>T(−∞)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>−0</td>
<td>Xi: T(dNaN)</td>
<td>T(+)</td>
<td>T(−0)</td>
<td>T(−0)</td>
<td>Xi: T(dNaN)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>Xi: T(dNaN)</td>
<td>T(−0)</td>
<td>T(−0)</td>
<td>T(+0)</td>
<td>T(+0)</td>
<td>Xi: T(dNaN)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>+Fn</td>
<td>T(−∞)</td>
<td>R(a·b)</td>
<td>T(−0)</td>
<td>T(+0)</td>
<td>R(a·b)</td>
<td>T(+)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>+∞</td>
<td>T(−∞)</td>
<td>T(−∞)</td>
<td>Xi: T(dNaN)</td>
<td>Xi: T(dNaN)</td>
<td>T(+∞)</td>
<td>T(+∞)</td>
<td>T(b)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>QNaN</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>T(a)</td>
<td>Xi: T(b*)</td>
</tr>
<tr>
<td>SNaN</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
<td>Xi: T(a*)</td>
</tr>
</tbody>
</table>

Explanation:
* The SNaN is converted to the corresponding QNaN before it is placed at the target operand location.
dNaN Default quiet NaN.
Fn Finite nonzero number (includes both denormalized and normalized).
R(v) Rounding and range action is performed on the value v. See [Figure 19-17 on page 19-21]
T(x) The value x is placed at the target operand location.
Xi: IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.

Figure 19-26. Results: MULTIPLY
MULTIPLY AND ADD

Mnemonic1  R₁,R₃,R₂  [RRF]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>///</th>
<th>R₃</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

Mnemonic1  Op Code  Operands
MAEBR  'B30E'  Short BFP
MADBR  'B31E'  Long BFP

Mnemonic2  R₁,R₃,Dz(X₂,B₂)  [RXF]

MULTIPLY AND SUBTRACT

Mnemonic1  R₁,R₃,R₂  [RRF]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>///</th>
<th>R₃</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

Mnemonic1  Op Code  Operands
MSEBR  'B30F'  Short BFP
MSDBR  'B31F'  Long BFP

Mnemonic2  R₁,R₃,Dz(X₂,B₂)  [RXF]

The third operand is multiplied by the second operand, and then the first operand is added to or subtracted from the product. The sum or difference is placed at the first-operand location. The MULTIPLY AND ADD and MULTIPLY AND SUBTRACT operations may be summarized as:

\[ \text{op₁} = \text{op₁} \cdot \text{op₂} \pm \text{op₁} \]

When the operands are numeric and finite, the third and second BFP operands are multiplied, forming an intermediate product, and the first operand is then added (or subtracted) algebraically to (or from) the intermediate product, forming an intermediate sum. The intermediate sum, if nonzero, is normalized and rounded to the operand format according to the current rounding mode and then placed at the first-operand location. The exponent and fraction of the intermediate product are maintained exactly; rounding and range checking occur only on the intermediate sum.

See Figure 19-27 on page 19-42 for a detailed description of the results of MULTIPLY AND ADD. The results of MULTIPLY AND SUBTRACT are the same, except that the first operand participates in the operation with its sign bit inverted.

**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:**
- Invalid operation
- Overflow
- Underflow
- Inexact

**Program Exceptions:**
- Access (fetch, operand 2 of MAEB, MADB, MSEB, MSDB)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)

**Programming Note:** MULTIPLY AND ADD and MULTIPLY AND SUBTRACT produce a precise intermediate result, and a single rounding operation is performed after the addition or subtraction. This definition is consistent with the RS/6000, and, in certain applications, can be used to great advantage, especially in algorithms used in math libraries.
<table>
<thead>
<tr>
<th>Third Operand &lt;br&gt; (a) Is</th>
<th>$\infty$</th>
<th>$-\infty$</th>
<th>$-F_n$</th>
<th>$-0$</th>
<th>$+0$</th>
<th>$+F_n$</th>
<th>$+\infty$</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$P(\infty)$</td>
<td>$P(\infty)$</td>
<td>Xi: $T(dNaN)$</td>
<td>Xi: $T(dNaN)$</td>
<td>$P(-\infty)$</td>
<td>$P(-\infty)$</td>
<td>$P(b)$</td>
<td>Xi: $T(b^*)$</td>
<td></td>
</tr>
<tr>
<td>$-F_n$</td>
<td>$P(\infty)$</td>
<td>$P(a\cdot b)$</td>
<td>$P(+0)$</td>
<td>$P(-0)$</td>
<td>$P(a\cdot b)$</td>
<td>$P(-\infty)$</td>
<td>$P(b)$</td>
<td>Xi: $T(b^*)$</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>Xi: $T(dNaN)$</td>
<td>$P(+0)$</td>
<td>$P(+0)$</td>
<td>$P(-0)$</td>
<td>$P(-0)$</td>
<td>Xi: $T(dNaN)$</td>
<td>$P(b)$</td>
<td>Xi: $T(b^*)$</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>Xi: $T(dNaN)$</td>
<td>$P(-0)$</td>
<td>$P(-0)$</td>
<td>$P(+0)$</td>
<td>$P(+0)$</td>
<td>Xi: $T(dNaN)$</td>
<td>$P(b)$</td>
<td>Xi: $T(b^*)$</td>
<td></td>
</tr>
<tr>
<td>$+F_n$</td>
<td>$P(-\infty)$</td>
<td>$P(a\cdot b)$</td>
<td>$P(-0)$</td>
<td>$P(+0)$</td>
<td>$P(a\cdot b)$</td>
<td>$P(+\infty)$</td>
<td>$P(b)$</td>
<td>Xi: $T(b^*)$</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$P(-\infty)$</td>
<td>$P(-\infty)$</td>
<td>Xi: $T(dNaN)$</td>
<td>Xi: $T(dNaN)$</td>
<td>$P(+\infty)$</td>
<td>$P(+\infty)$</td>
<td>$P(b)$</td>
<td>Xi: $T(b^*)$</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>$P(a)$</td>
<td>$P(a)$</td>
<td>$P(a)$</td>
<td>$P(a)$</td>
<td>$P(a)$</td>
<td>$P(a)$</td>
<td>$P(a)$</td>
<td>Xi: $T(b^*)$</td>
<td></td>
</tr>
<tr>
<td>SNaN</td>
<td>Xi: $T(a^*)$</td>
<td>Xi: $T(a^*)$</td>
<td>Xi: $T(a^*)$</td>
<td>Xi: $T(a^*)$</td>
<td>Xi: $T(a^*)$</td>
<td>Xi: $T(a^*)$</td>
<td>Xi: $T(a^*)$</td>
<td>Xi: $T(a^*)$</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 19-27 (Part 1 of 2). Results: MULTIPLY AND ADD*
Figure 19-27 (Part 2 of 2). Results: MULTIPLY AND ADD

<table>
<thead>
<tr>
<th>Value from Part 1 (p) Is</th>
<th>–∞</th>
<th>–Fn</th>
<th>–0</th>
<th>+0</th>
<th>+Fn</th>
<th>+∞</th>
<th>QNaN</th>
<th>SNaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>–∞</td>
<td>T(–∞)</td>
<td>T(–∞)</td>
<td>T(–∞)</td>
<td>T(–∞)</td>
<td>Xi: T(dNaN)</td>
<td>T(c)</td>
<td>Xi: T(c*)</td>
<td></td>
</tr>
<tr>
<td>–Fn</td>
<td>T(–∞)</td>
<td>R(p+c)</td>
<td>R(p)</td>
<td>R(p)</td>
<td>R(p+c)</td>
<td>T(+∞)</td>
<td>T(c)</td>
<td>Xi: T(c*)</td>
</tr>
<tr>
<td>–0</td>
<td>T(–∞)</td>
<td>R(c)</td>
<td>T(–0)</td>
<td>Rezd</td>
<td>R(c)</td>
<td>T(+∞)</td>
<td>T(c)</td>
<td>Xi: T(c*)</td>
</tr>
<tr>
<td>+0</td>
<td>T(–∞)</td>
<td>R(c)</td>
<td>Rezd</td>
<td>T(+0)</td>
<td>R(c)</td>
<td>T(+∞)</td>
<td>T(c)</td>
<td>Xi: T(c*)</td>
</tr>
<tr>
<td>+Fn</td>
<td>T(–∞)</td>
<td>R(p+c)</td>
<td>R(p)</td>
<td>R(p)</td>
<td>R(p+c)</td>
<td>T(+∞)</td>
<td>T(c)</td>
<td>Xi: T(c*)</td>
</tr>
<tr>
<td>+∞</td>
<td>Xi: T(dNaN)</td>
<td>T(+∞)</td>
<td>T(+∞)</td>
<td>T(+∞)</td>
<td>T(+∞)</td>
<td>T(c)</td>
<td>Xi: T(c*)</td>
<td></td>
</tr>
<tr>
<td>QNaN</td>
<td>T(p)</td>
<td>T(p)</td>
<td>T(p)</td>
<td>T(p)</td>
<td>T(p)</td>
<td>T(p)</td>
<td>T(p)</td>
<td>Xi: T(c*)</td>
</tr>
</tbody>
</table>

Explanation:

* The SNaN is converted to the corresponding QNaN before it is placed at the target operand location.

dNaN Default quiet NaN.

Fn Finite nonzero number (includes both denormalized and normalized).

P(x) The value x is passed to Part 2 of this figure.

R(v) Rounding and range action is performed on the value v. See Figure 19-17 on page 19-21.

Rezd Exact zero-difference result. See Figure 19-17 on page 19-21.

T(x) The value x is placed at the target operand location.

Xi: IEEE invalid-operation exception. The results shown are produced only when FPC 0.0 is zero.

IEEE Exception Conditions: None.

Program Exceptions:

* Data with DXC 2, BFP instruction
* Operation (if the BFP facility is not installed)
* Specification

SET ROUNDBING MODE

<table>
<thead>
<tr>
<th>SRNM</th>
<th>D2 (B2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>'B299'</td>
<td>B2</td>
</tr>
</tbody>
</table>

The rounding-mode bits are set from the second-operand address.

The second-operand address is not used to address data; instead, the rounding-mode bits in the FPC register are set with bits 62 and 63 of the address.

Bits other than 62 and 63 of the second-operand address are ignored.

**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:** None.

**Program Exceptions:**
- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)

### SQUARE ROOT

Mnemonic1 | R₁,R₂ | [RRE]
--- | --- | ---

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Mnemonic1 | Op Code | Operands
--- | --- | ---
SQEB | 'B314' | Short BFP
SQDB | 'B315' | Long BFP
SXBR | 'B316' | Extended BFP

Mnemonic2 | R₁,Dₓ(X₂,B₂) | [RXE]
--- | --- | ---

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>Dₓ</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Mnemonic2 | Op Code | Operands
--- | --- | ---
SQEB | 'ED14' | Short BFP
SQDB | 'ED15' | Long BFP

The square root of the second operand is placed at the first-operand location.

The result rounded according to the current rounding mode is placed at the first-operand location.

If the second operand is a finite positive number, the result is the square root of that number with a plus sign. If the operand is a zero of either sign, the result is a zero of the same sign. If the operand is +∞, the result is +∞.

If the second operand is less than zero, an IEEE-invalid-operation condition is recognized.

See Figure 19-21 on page 19-27 for a detailed description of the results of this instruction.

For SQXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:** None.

**Program Exceptions:**
- Access (fetch, operand 2 of SQEB and SQDB only)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (SQXBR only)

### STORE FPC

STFPC | Dₓ(B₂) | [S]
--- | --- | ---

<table>
<thead>
<tr>
<th>'B29C'</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The contents of the FPC (floating-point-control) register are placed in storage at the second-operand location.

The operand is four bytes in length. All 32 bits of the FPC register are stored.

**Condition Code:** The code remains unchanged.

**IEEE Exception Conditions:** None.

**Program Exceptions:**
- Access (store, operand 2)
- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)

### SUBTRACT

Mnemonic1 | R₁,R₂ | [RRE]
--- | --- | ---

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

Mnemonic1 | Op Code | Operands
--- | --- | ---
SEBR | 'B30B' | Short BFP
SOBR | 'B31B' | Long BFP
SXBR | 'B34B' | Extended BFP
Mnemonic2  R1,Dz(Xz,Bz)  [RXE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>Xz</th>
<th>Bz</th>
<th>Dz</th>
<th>--------/</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Mnemonic2  Op Code  Operands
SEB  'ED0B'  Short BFP
SDB  'ED1B'  Long BFP

The second operand is subtracted from the first operand, and the difference is placed at the first-operand location.

The execution of SUBTRACT is identical to that of ADD, except that the second operand participates in the operation with its sign bit inverted. See Figure 19-16 on page 19-20 for the detailed results of ADD.

For SXBR, the R fields must designate valid floating-point-register pairs; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0  Result is zero  
1  Result is less than zero  
2  Result is greater than zero  
3  Result is a NaN

**IEEE Exception Conditions:**

- Invalid operation
- Overflow
- Underflow
- Inexact

**Program Exceptions:**

- Access (fetch, operand 2 of SEB and SDB only)
- Data with DXC 2, BFP instruction
- Data with DXC for IEEE exception condition
- Operation (if the BFP facility is not installed)
- Specification (SXBR only)

---

**TEST DATA CLASS**

Mnemonic  R1,Dz(Xz,Bz)  [RXE]

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>Xz</th>
<th>Bz</th>
<th>Dz</th>
<th>--------/</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>32</td>
<td>40</td>
</tr>
</tbody>
</table>

Mnemonic  Op Code  Operands
TCEB  'ED10'  Short BFP
TCDB  'ED11'  Long BFP
TCXB  'ED12'  Extended BFP

The class and sign of the first operand are examined to select one bit from the second-operand address. Condition code 0 or 1 is set according to whether the selected bit is zero or one, respectively.

The second-operand address is not used to address data; instead, the rightmost 12 bits of the address, bits 20-31, are used to specify 12 combinations of operand class and sign. Bits 0-19 of the second-operand address are ignored.

As shown in Figure 19-28, BFP operands are divided into six classes: zero, normalized number, denormalized number, infinity, quiet NaN, and signaling NaN.

<table>
<thead>
<tr>
<th>BFP Operand Class</th>
<th>Bit Used when Sign Is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>+ 20</td>
</tr>
<tr>
<td>Normalized number</td>
<td>22</td>
</tr>
<tr>
<td>Denormalized number</td>
<td>24</td>
</tr>
<tr>
<td>Infinity</td>
<td>26</td>
</tr>
<tr>
<td>Quiet NaN</td>
<td>28</td>
</tr>
<tr>
<td>Signaling NaN</td>
<td>30</td>
</tr>
</tbody>
</table>

*Figure 19-28. Second-Operand-Address Bits for TEST DATA CLASS*

One or more of the second-operand-address bits may be set to one. If the second-operand-address bit corresponding to the class and sign of the first operand is one, condition code 1 is set; otherwise, condition code 0 is set.

Operands, including SNaNs and QNaNs, are examined without causing an arithmetic exception.
For TCXB, the $R_1$ field must designate a valid floating-point-register pair; otherwise, a specification exception is recognized.

**Resulting Condition Code:**

0  Selected bit is 0 (no match)
1  Selected bit is 1 (match)
2  --
3  --

**IEEE Exception Conditions:** None.

**Program Exceptions:**

- Data with DXC 2, BFP instruction
- Operation (if the BFP facility is not installed)
- Specification (TCXB only)

**Programming Note:** TEST DATA CLASS provides a way to test an operand without risk of an exception or setting the IEEE flags.
## Appendix A. Number Representation and Instruction-Use Examples

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Representation</td>
<td>A-2</td>
</tr>
<tr>
<td>Binary Integers</td>
<td>A-2</td>
</tr>
<tr>
<td>Signed Binary Integers</td>
<td>A-2</td>
</tr>
<tr>
<td>Unsigned Binary Integers</td>
<td>A-3</td>
</tr>
<tr>
<td>Decimal Integers</td>
<td>A-4</td>
</tr>
<tr>
<td>Hexadecimal-Floating-Point Numbers</td>
<td>A-5</td>
</tr>
<tr>
<td>Conversion Example</td>
<td>A-6</td>
</tr>
<tr>
<td>Instruction-Use Examples</td>
<td>A-6</td>
</tr>
<tr>
<td>Machine Format</td>
<td>A-7</td>
</tr>
<tr>
<td>Assembler-Language Format</td>
<td>A-7</td>
</tr>
<tr>
<td>Addressing Mode in Examples</td>
<td>A-7</td>
</tr>
<tr>
<td>General Instructions</td>
<td>A-7</td>
</tr>
<tr>
<td>ADD HALFWORD (AH)</td>
<td>A-7</td>
</tr>
<tr>
<td>AND (N, NC, NI, NR)</td>
<td>A-8</td>
</tr>
<tr>
<td>NI Example</td>
<td>A-8</td>
</tr>
<tr>
<td>Linkage Instructions (BAL, BALR, BAS, BASR, BASSM, BSM)</td>
<td>A-8</td>
</tr>
<tr>
<td>Other BALR and BASR Examples</td>
<td>A-9</td>
</tr>
<tr>
<td>BRANCH AND STACK (BAKR)</td>
<td>A-10</td>
</tr>
<tr>
<td>BAKR Example 1</td>
<td>A-10</td>
</tr>
<tr>
<td>BAKR Example 2</td>
<td>A-11</td>
</tr>
<tr>
<td>BAKR Example 3</td>
<td>A-11</td>
</tr>
<tr>
<td>BRANCH ON CONDITION (BC, BCR)</td>
<td>A-11</td>
</tr>
<tr>
<td>BRANCH ON COUNT (BCT, BCTR)</td>
<td>A-12</td>
</tr>
<tr>
<td>BRANCH ON INDEX HIGH (BXH)</td>
<td>A-12</td>
</tr>
<tr>
<td>BXH Example 1</td>
<td>A-12</td>
</tr>
<tr>
<td>BXH Example 2</td>
<td>A-13</td>
</tr>
<tr>
<td>BRANCH ON INDEX LOW OR EQUAL (BXLE)</td>
<td>A-13</td>
</tr>
<tr>
<td>BXLE Example 1</td>
<td>A-13</td>
</tr>
<tr>
<td>BXLE Example 2</td>
<td>A-14</td>
</tr>
<tr>
<td>COMPARE AND FORM CODEWORD (CFC)</td>
<td>A-14</td>
</tr>
<tr>
<td>COMPARE HALFWORD (CH)</td>
<td>A-14</td>
</tr>
<tr>
<td>COMPARE LOGICAL (CL, CLC, CLI, CLR)</td>
<td>A-14</td>
</tr>
<tr>
<td>CLC Example</td>
<td>A-14</td>
</tr>
<tr>
<td>CLI Example</td>
<td>A-15</td>
</tr>
<tr>
<td>CLR Example</td>
<td>A-15</td>
</tr>
<tr>
<td>COMPARE LOGICAL CHARACTERS UNDER MASK</td>
<td>A-15</td>
</tr>
<tr>
<td>UNDER MASK (CLM)</td>
<td>A-15</td>
</tr>
<tr>
<td>COMPARE LOGICAL LONG (CLCL)</td>
<td>A-16</td>
</tr>
<tr>
<td>COMPARE LOGICAL STRING (CLST)</td>
<td>A-17</td>
</tr>
<tr>
<td>CONVERT TO BINARY (CVB)</td>
<td>A-18</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (CVD)</td>
<td>A-18</td>
</tr>
<tr>
<td>DIVIDE (D, DR)</td>
<td>A-19</td>
</tr>
<tr>
<td>EXCLUSIVE OR (X, XC, XI, XR)</td>
<td>A-19</td>
</tr>
<tr>
<td>XC Example</td>
<td>A-19</td>
</tr>
<tr>
<td>XI Example</td>
<td>A-20</td>
</tr>
<tr>
<td>EXECUTE (EX)</td>
<td>A-21</td>
</tr>
<tr>
<td>INSERT CHARACTERS UNDER MASK (ICM)</td>
<td>A-21</td>
</tr>
<tr>
<td>LOAD (L, LR)</td>
<td>A-22</td>
</tr>
<tr>
<td>LOAD ADDRESS (LA)</td>
<td>A-22</td>
</tr>
<tr>
<td>LOAD HALFWORD (LH)</td>
<td>A-23</td>
</tr>
<tr>
<td>MOVE (MVCWORD, MVI)</td>
<td>A-23</td>
</tr>
<tr>
<td>MVC Example</td>
<td>A-23</td>
</tr>
<tr>
<td>MVI Example</td>
<td>A-24</td>
</tr>
<tr>
<td>MOVE INVERSE (MVCIN)</td>
<td>A-24</td>
</tr>
<tr>
<td>MOVE LONG (MVCL)</td>
<td>A-25</td>
</tr>
<tr>
<td>MOVE NUMERICS (MVN)</td>
<td>A-25</td>
</tr>
<tr>
<td>MOVE STRING (MVST)</td>
<td>A-26</td>
</tr>
<tr>
<td>MOVE WITH OFFSET (MVO)</td>
<td>A-26</td>
</tr>
<tr>
<td>MOVE ZONES (MVZ)</td>
<td>A-27</td>
</tr>
<tr>
<td>MULTIPLY (M, MR)</td>
<td>A-27</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD (MH)</td>
<td>A-27</td>
</tr>
<tr>
<td>OR (O, OC, OI, OR)</td>
<td>A-28</td>
</tr>
<tr>
<td>OI Example</td>
<td>A-28</td>
</tr>
<tr>
<td>PACK (PACK)</td>
<td>A-28</td>
</tr>
<tr>
<td>SEARCH STRING (SRST)</td>
<td>A-29</td>
</tr>
<tr>
<td>SRST Example 1</td>
<td>A-29</td>
</tr>
<tr>
<td>SRST Example 2</td>
<td>A-29</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE (SLDA)</td>
<td>A-29</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE (SLA)</td>
<td>A-30</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (STCM)</td>
<td>A-30</td>
</tr>
<tr>
<td>STORE MULTIPLE (STM)</td>
<td>A-30</td>
</tr>
<tr>
<td>TEST UNDER MASK (TM)</td>
<td>A-31</td>
</tr>
<tr>
<td>TRANSLATE (TR)</td>
<td>A-31</td>
</tr>
<tr>
<td>TRANSLATE AND TEST (TRT)</td>
<td>A-32</td>
</tr>
<tr>
<td>UNPACK (UNPK)</td>
<td>A-33</td>
</tr>
<tr>
<td>UPDATE TREE (UPT)</td>
<td>A-34</td>
</tr>
<tr>
<td>Decimal Instructions</td>
<td>A-34</td>
</tr>
<tr>
<td>ADD DECIMAL (AP)</td>
<td>A-34</td>
</tr>
<tr>
<td>COMPARE DECIMAL (CP)</td>
<td>A-34</td>
</tr>
<tr>
<td>DIVIDE DECIMAL (DP)</td>
<td>A-34</td>
</tr>
<tr>
<td>EDIT (ED)</td>
<td>A-35</td>
</tr>
<tr>
<td>EDIT AND MARK (EDMK)</td>
<td>A-36</td>
</tr>
<tr>
<td>MULTIPLY DECIMAL (MP)</td>
<td>A-36</td>
</tr>
<tr>
<td>SHIFT AND ROUND DECIMAL (SRP)</td>
<td>A-37</td>
</tr>
<tr>
<td>Decimal Left Shift</td>
<td>A-37</td>
</tr>
<tr>
<td>Decimal Right Shift</td>
<td>A-37</td>
</tr>
<tr>
<td>Decimal Right Shift and Round</td>
<td>A-38</td>
</tr>
<tr>
<td>Multiplying by a Variable Power of 10</td>
<td>A-38</td>
</tr>
<tr>
<td>ZERO AND ADD (ZAP)</td>
<td>A-38</td>
</tr>
<tr>
<td>Hexadecimal-Floating-Point Instructions</td>
<td>A-39</td>
</tr>
<tr>
<td>ADD NORMALIZED (AD, ADR, AE, AER, AXR)</td>
<td>A-39</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1990-2004
Number Representation

Binary Integers

Signed Binary Integers
Signed binary integers are most commonly represented as halfwords (16 bits) or words (32 bits). In both lengths, the leftmost bit (bit 0) is the sign of the number. The remaining bits (bits 1-15 for halfwords and 1-31 for words) are used to specify the magnitude of the number. Binary integers are also referred to as fixed-point numbers, because the radix point (binary point) is considered to be fixed at the right, and any scaling is done by the programmer.

Positive binary integers are in true binary notation with a zero sign bit. Negative binary integers are in two's-complement notation with a one bit in the sign position. In all cases, the bits between the sign bit and the leftmost significant bit of the integer are the same as the sign bit (that is, all zeros for positive numbers, all ones for negative numbers).

Negative binary integers are formed in two's-complement notation by inverting each bit of the positive binary integer and adding one. As an example using the halfword format, the binary number with the decimal value +26 is made negative (-26) in the following manner:

$$
\begin{align*}
+26 & = 0000000001101111 \\
\text{Invert} & = 1111111000000000 \\
\text{Add} 1 & = 1111111111100101 \\
\end{align*}
$$

(S is the sign bit.)

This is equivalent to subtracting the number:

$$
\begin{align*}
0000000000011010 \\
\text{from} & = 1111111111100101 \\
\end{align*}
$$

Negative binary integers are changed to positive in the same manner.

The following addition examples illustrate two's-complement arithmetic and overflow conditions. Only eight bit positions are used.

1. \[ +57 = 00111001 \\
    +35 = 00100011 \]
   \[ +92 = 01011100 \]

2. \[ +57 = 00111001 \\
    -35 = 11011101 \]
   \[ +22 = 00010110 \] No overflow — carry into leftmost position and carry out
3. \[+35 = 0010\ 0011\]
\[-57 = 1100\ 0111\]
\[-22 = 1110\ 1010\]
Sign change only — no carry into leftmost position and no carry out

4. \[-57 = 1100\ 0111\]
\[-35 = 1101\ 1101\]
\[-92 = 1010\ 0100\]
No overflow — carry into leftmost position and carry out

5. \[+57 = 0011\ 1001\]
\[+92 = 1001\ 1100\]
\[+149 = 1001\ 1011\]
*Overflow — carry into leftmost position, no carry out

6. \[-57 = 1100\ 0111\]
\[-92 = 1010\ 0100\]
\[-149 = 0110\ 1011\]
*Overflow — no carry into leftmost position but carry out

The presence or absence of an overflow condition may be recognized from the carries:

- There is no overflow:

\[+35 = 0010\ 0011\]
\[-57 = 1100\ 0111\]
\[-22 = 1110\ 1010\]

- There is an overflow:

\[-57 = 1100\ 0111\]
\[-35 = 1101\ 1101\]
\[-92 = 1010\ 0100\]
*Carry out of leftmost position

The following are 16-bit signed binary integers. The first is the maximum positive 16-bit binary integer. The last is the maximum negative 16-bit binary integer (the negative 16-bit binary integer with the greatest absolute value).

\[2^{15} - 1 = 32,767 = 0\ 111\ 1111\ 1111\ 1111\]
\[2^0 = 1 = 0\ 000\ 0000\ 0000\ 0000\]
\[0 = 0 = 0\ 000\ 0000\ 0000\ 0000\]
\[-2^0 = -1 = 1\ 111\ 1111\ 1111\ 1111\]
\[-2^{15} = -32,768 = 1\ 000\ 0000\ 0000\ 0000\]

Figure A-1 illustrates several 32-bit signed binary integers arranged in descending order. The first is the maximum positive binary integer that can be represented by 32 bits, and the last is the maximum negative binary integer that can be represented by 32 bits.

\[2^{31} - 1 = 2\ 147\ 483\ 647 = 0\ 111\ 1111\ 1111\ 1111\]
\[2^{16} = 65\ 536 = 0\ 000\ 0000\ 0000\ 0001\]
\[2^0 = 1 = 0\ 000\ 0000\ 0000\ 0000\]
\[0 = 0 = 0\ 000\ 0000\ 0000\ 0000\]
\[-2^0 = -1 = 1\ 111\ 1111\ 1111\ 1111\]
\[-2^{16} = -65\ 536 = 1\ 111\ 1111\ 1111\ 1111\]
\[-2^{31} + 1 = -2\ 147\ 483\ 647 = 1\ 000\ 0000\ 0000\ 0000\]
\[-2^{31} = -2\ 147\ 483\ 648 = 1\ 000\ 0000\ 0000\ 0000\]

**Figure A-1. 32-Bit Signed Binary Integers**

**Unsigned Binary Integers**

Certain instructions, such as ADD LOGICAL, treat binary integers as unsigned rather than signed. Unsigned binary integers have the same format as signed binary integers, except that the leftmost bit is interpreted as another numeric bit rather than a sign bit. There is no complement notation because all unsigned binary integers are considered positive.

The following examples illustrate the addition of unsigned binary integers. Only eight bit positions are used. The examples are numbered the same as the corresponding examples for signed binary integers.

1. \[57 = 0011\ 1001\]
\[35 = 0010\ 0011\]
\[92 = 0101\ 1100\]

2. \[57 = 0011\ 1001\]
\[221 = 1101\ 1101\]
\[278 = 0001\ 0110\]
*Carry out of leftmost position*
Decimal Integers

Decimal integers consist of one or more decimal digits and a sign. Each digit and the sign are represented by a 4-bit code. The decimal digits are in binary-coded decimal (BCD) form, with the values 0-9 encoded as 0000-1001. The sign is usually represented as 1100 (C hex) for plus and 1101 (D hex) for minus. These are the preferred sign codes, which are generated by the machine for the results of decimal-arithmetic operations. There are also several alternate sign codes (1010, 1110, and 1111 for plus; 1011 for minus). The alternate sign codes are accepted by the machine as valid in source operands but are not generated for results.

Decimal integers may have different lengths, from one to 16 bytes. There are two decimal formats: packed and zoned. In the packed format, each byte contains two decimal digits, except for the rightmost byte, which contains the sign code in the right half. For decimal arithmetic, the number of decimal digits in the packed format can vary from one to 31. Because decimal integers must consist of whole bytes and there must be a sign code on the right, the number of decimal digits is always odd. If an even number of significant digits is desired, a leading zero must be inserted on the left.

In the zoned format, each byte consists of a decimal digit on the right and the zone code 1111 (F hex) on the left, except for the rightmost byte where the sign code replaces the zone code. Thus, a decimal integer in the zoned format can have from one to 16 digits. The zoned format may be used directly for input and output in the extended binary-coded-decimal interchange code (EBCDIC), except that the sign must be separated from the rightmost digit and handled as a separate character. For positive (unsigned) numbers, however, the sign can simply be represented by the zone code of the rightmost digit because the zone code is one of the acceptable alternate codes for plus.

In either format, negative decimal integers are represented in true notation with a separate sign. As for binary integers, the radix point (decimal point) of decimal integers is considered to be fixed at the right, and any scaling is done by the programmer.

The following are some examples of decimal integers shown in hexadecimal notation:

\[
\begin{align*}
2^{32} - 1 &= 4294967295 = 11111111111111111111111111111111 \\
2^{31} &= 2147483648 = 10000000000000000000000000000000 \\
2^{31} - 1 &= 2147483647 = 11111111111111111111111111111111 \\
2^{16} &= 65536 = 00000000000000101000000000000000 \\
2^8 &= 1 = 00000000000000000000000000000001 \\
0 &= 00000000000000000000000000000000 \\
\end{align*}
\]

Figure A-2 illustrates several 32-bit unsigned binary integers arranged in descending order.
Under some circumstances, a zero with a minus sign (negative zero) is produced. For example, the multiplicand:

\[
00 \ 12 \ 3D \ (-123)
\]
times the multiplier:

\[
0C \ (+0)
\]
generates the product:

\[
00 \ 00 \ 00 \ (-0)
\]
because the product sign follows the algebraic rule of signs even when the value is zero. A negative zero, however, is equivalent to a positive zero in that they compare equal in a decimal comparison.

**Hexadecimal-Floating-Point Numbers**

A hexadecimal-floating-point (HFP) number is expressed as a hexadecimal fraction multiplied by a separate power of 16. The term floating point indicates that the placement, of the radix (hexadecimal) point, or scaling, is automatically maintained by the machine.

The part of an HFP number which represents the significant digits of the number is called the fraction. A second part specifies the power (exponent) to which 16 is raised and indicates the location of the radix point of the number. The fraction and exponent may be represented by 32 bits (short format), 64 bits (long format), or 128 bits (extended format).
More precisely,

In the short format:
\[ 16^{-65} \leq M \leq (1 - 16^{-6}) \times 16^{63} \]
In the long format:
\[ 16^{-65} \leq M \leq (1 - 16^{-14}) \times 16^{63} \]
In the extended format:
\[ 16^{-65} \leq M \leq (1 - 16^{-28}) \times 16^{63} \]

Within a given fraction length (6, 14, or 28 digits), an HFP operation will provide the greatest precision if the fraction is normalized. A fraction is normalized when the leftmost digit (bit positions 8, 9, 10, and 11) is nonzero. It is unnormalized if the leftmost digit contains all zeros.

If normalization of the operand is desired, the HFP instructions that provide automatic normalization are used. This automatic normalization is accomplished by left-shifting the fraction (four bits per shift) until a nonzero digit occupies the leftmost digit position. The characteristic is reduced by one for each digit shifted.

Figure A-3 illustrates sample normalized short HFP numbers. The last two numbers represent the smallest and the largest positive normalized numbers.

<table>
<thead>
<tr>
<th>Number</th>
<th>Hexadecimal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>+1/16x16^1</td>
</tr>
<tr>
<td>0.5</td>
<td>+8/16x16^0</td>
</tr>
<tr>
<td>1/64</td>
<td>+4/16x16^-1</td>
</tr>
<tr>
<td>0.0</td>
<td>+0 x16^-64</td>
</tr>
<tr>
<td>-15.0</td>
<td>-15/16x16^1</td>
</tr>
<tr>
<td>5.4x10^-79</td>
<td>+1/16x16^-64</td>
</tr>
<tr>
<td>7.2x10^-75</td>
<td>(1-16^-6)x16^63</td>
</tr>
</tbody>
</table>

Figure A-3. Normalized Short Hexadecimal-Floating-Point Numbers

Conversion Example

Convert the decimal number 59.25 to a short HFP number. (In another appendix are tables for the conversion of hexadecimal and decimal integers and fractions.)

1. The number is separated into a decimal integer and a decimal fraction.
   \[ 59.25 = 59 + 0.25 \]

2. The decimal integer is converted to its hexadecimal representation.
   \[ 59_{10} = 3B_{16} \]

3. The decimal fraction is converted to its hexadecimal representation.
   \[ 0.25_{10} = 0.4_{16} \]

4. The integral and fractional parts are combined and expressed as a fraction times a power of 16 (exponent).
   \[ 38.4_{16} = 0.384_{16} \times 16^2 \]

5. The characteristic is developed from the exponent and converted to binary.
   \[ \text{base + exponent} = \text{characteristic} \]
   \[ 64 + 2 = 66 = 1000010 \]

6. The fraction is converted to binary and grouped hexadecimally.
   \[ .384_{16} = .0011 1011 0100 \]

7. The characteristic and the fraction are stored in the short format. The sign position contains the sign of the fraction.

<table>
<thead>
<tr>
<th>S Char</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000010 0011 1011 0100 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

Examples of instruction sequences that may be used to convert between signed binary integers and HFP numbers are shown in "Hexadecimal-Floating-Point-Number Conversion" on page A-42.

Instruction-Use Examples

The following examples illustrate the use of many of the unprivileged instructions. Before studying one of these examples, the reader should consult the instruction description.

The instruction-use examples are written principally for assembler-language programmers, to be used in conjunction with the appropriate assembler-language publications.
Most examples present one particular instruction, both as it is written in an assembler-language statement and as it appears when assembled in storage (machine format).

**Machine Format**

All machine-format values are given in hexadecimal notation unless otherwise specified. Storage addresses are also given in hexadecimal. Hexadecimal operands are shown converted into binary, decimal, or both if such conversion helps to clarify the example for the reader.

**Assembler-Language Format**

In assembler-language statements, registers and lengths are presented in decimal. Displacements, immediate operands, and masks may be shown in decimal, hexadecimal, or binary notation; for example, 12, X'C', and B'1100' represent the same value. Whenever the value in a register or storage location is referred to as "not significant," this value is replaced during the execution of the instruction.

When SS-format instructions are written in the assembler language, lengths are given as the total number of bytes in the field. This differs from the machine definition, in which the length field specifies the number of bytes to be added to the field address to obtain the address of the last byte of the field. Thus, the machine length is one less than the assembler-language length. The assembler program automatically subtracts one from the length specified when the instruction is assembled.

In some of the examples, symbolic addresses are used in order to simplify the examples. In assembler-language statements, a symbolic address is represented as a mnemonic term written in all capitals, such as FLAGS, which may denote the address of a storage location containing data or program-control information. When symbolic addresses are used, the assembler supplies actual base and displacement values according to the programmer's specifications. Therefore, the actual values for base and displacement are not shown in the assembler-language format or in the machine-language format. For assembler-language formats, in the labels that designate instruction fields, the letter "S" is used to indicate the combination of base and displacement fields for an operand address. (For example, S2 represents the combination of B2 and D2.) In the machine-language format, the base and displacement address components are shown as asterisks (****).

**Addressing Mode in Examples**

Except where otherwise specified, the examples assume the 24-bit addressing mode.

---

**General Instructions**

(See Chapter 7, "General Instructions" for a complete description of the general instructions.)

**ADD HALFWORD (AH)**

The ADD HALFWORD instruction algebraically adds the contents of a two-byte field in storage to the contents of a register. The storage operand is expanded to 32 bits after it is fetched and before it is used in the add operation. The expansion consists in propagating the leftmost (sign) bit 16 positions to the left. For example, assume that the contents of storage locations 2000-2001 are to be added to register 5. Initially:

- Register 5 contains 00 00 00 19 = 25.
- Register 12 contains 00 00 18 00.
- Register 13 contains 00 00 01 50.

The format of the required instruction is:

<table>
<thead>
<tr>
<th>Machine Format</th>
<th>Assembler Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Op Code</strong></td>
<td>**AH 5,X'6B/<em>j/<em>j</em></em> (13,12)</td>
</tr>
<tr>
<td><strong>R</strong></td>
<td></td>
</tr>
<tr>
<td><strong>X</strong></td>
<td></td>
</tr>
<tr>
<td><strong>B</strong></td>
<td></td>
</tr>
<tr>
<td><strong>D</strong></td>
<td></td>
</tr>
</tbody>
</table>

After the instruction is executed, register 5 contains 00 00 00 17 = 23. Condition code 2 is set to indicate a result greater than zero.
AND (N, NC, NI, NR)

When the Boolean operator AND is applied to two bits, the result is one when both bits are one; otherwise, the result is zero. When two bytes are ANDed, each pair of bits is handled separately; there is no connection from one bit position to another. The following is an example of ANDing two bytes:

First-operand byte: \(0011\ 0101\)
Second-operand byte: \(0101\ 1100\)

Result byte: \(0001\ 0100\)

NI Example

A frequent use of the AND instruction is to set a particular bit to zero. For example, assume that storage location 4891 contains \(0100\ 0011\). To set the rightmost bit of this byte to zero without affecting the other bits, the following instruction can be used (assume that register 8 contains 00 00 48 90):

Machine Format

\[
\begin{array}{c|c|c|c}
\text{Op Code} & I_z & B_1 & D_1 \\
\hline
94 & FE & 8 & 001
\end{array}
\]

Assembler Format

\[
\text{NI 1(8),X'FE'}
\]

When this instruction is executed, the byte in storage is ANDed with the immediate byte (the \(I_z\) field of the instruction):

Location 4891: \(0100\ 0011\)
Immediate byte: \(1111\ 1110\)

Result: \(0100\ 0000\)

The resulting byte, with bit 7 set to zero, is stored back in location 4891. Condition code 1 is set.

Linkage Instructions (BAL, BALR, BAS, BASR, BASSM, BSM)

Four unprivileged instructions (BRANCH AND LINK, BRANCH AND SAVE, BRANCH AND SAVE AND SET MODE, and BRANCH AND SET MODE) are available, together with the unconditional branch (BRANCH ON CONDITION with a mask of 15), to provide linkage between subroutines. BRANCH AND LINK (BAL or BALR) is provided primarily for compatibility with programs written for System/370; BRANCH AND SAVE (BAS or BASR) is recommended instead for programs which are to be executed using ESA/370. The instructions BRANCH AND SAVE AND SET MODE (BASSM) and BRANCH AND SET MODE (BSM) provide subroutine linkage together with switching between the 24-bit and the 31-bit addressing modes. The use of these instructions is discussed in a programming note at the end of “Subroutine Linkage without the Linkage Stack.” (See also the semiprivileged instruction BRANCH AND STACK.)

The following example compares the operation of these instructions and of the unconditional-branch instruction BRANCH ON CONDITION (BC or BCR with a mask of 15). Assume that each instruction in turn is located at the current instruction address, ready to be executed next. For the first set of examples, the addressing-mode bit, PSW bit 32, is initially zero (24-bit addressing in effect). For the second set, PSW bit 32 is initially one (31-bit addressing). Assume also that general register 5 is to receive the linkage information, and that general register 6 contains the branch address.

The format of the BALR instruction is:

Machine Format

\[
\begin{array}{c|c|c|c}
\text{Op Code} & R_1 & R_2 \\
\hline
05 & 5 & 6
\end{array}
\]

Assembler Format

\[
\text{BALR 5,6}
\]

The other linkage instructions in the RR format have the same format but different op codes:

\[
\begin{align*}
\text{BASR} & \ 0D \\
\text{BASSM} & \ 0C \\
\text{BSM} & \ 0B
\end{align*}
\]

For comparison with the RR-format instructions, the results of two RX-format instructions are also shown.

The format of the BAL instruction is:
Machine Format
Op Code R₁ X₂ B₂ D₂

| 45 | 5 | 0 | 6 | 000 |

Assembler Format
Op Code R₁,D₂(X₂,B₂)

BAL 5,0(0,6)

The BAS instruction has the same format, but the op code is 4D.

The BCR instruction specifies only one register:

Machine Format
Op Code M₁ R₂

| 07 | F | 6 |

Assembler Format
Op Code M₁,R₂

BCR 15,6

Assume that:

- Register 5 contains BB BB BB BB.
- Register 6 contains 82 46 8A CE.
- PSW bits 32-63 contain 00 00 10 D6 (for 24-bit addressing).
- PSW bits 32-63 contain 80 00 10 D6 (for 31-bit addressing).
- Condition code is 01₂.
- Program mask is 1100₂.

The effect of executing each instruction in turn is as follows:

### 24-Bit Mode Initially

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 5</th>
<th>PSW (32-63)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>BB BB BB BB</td>
<td>00 00 10 D6</td>
</tr>
<tr>
<td>BCR 15,6</td>
<td>BB BB BB BB</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>BAL 5,0(0,6)</td>
<td>9C 00 10 DA</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>BAS 5,0(0,6)</td>
<td>00 00 10 DA</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>BALR 5,6</td>
<td>5C 00 10 D8</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>BASR 5,6</td>
<td>00 00 10 D8</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>BASSM 5,6</td>
<td>00 00 10 D8</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>BSM 5,6</td>
<td>3B BB BB BB</td>
<td>82 46 8A CE</td>
</tr>
</tbody>
</table>

### 31-Bit Mode Initially

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 5</th>
<th>PSW (32-63)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>BB BB BB BB</td>
<td>80 00 10 D6</td>
</tr>
<tr>
<td>BCR 15,6</td>
<td>BB BB BB BB</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>BAL 5,0(0,6)</td>
<td>80 00 10 DA</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>BAS 5,0(0,6)</td>
<td>80 00 10 DA</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>BALR 5,6</td>
<td>80 00 10 D8</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>BASR 5,6</td>
<td>80 00 10 D8</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>BASSM 5,6</td>
<td>80 00 10 D8</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>BSM 5,6</td>
<td>BB BB BB BB</td>
<td>82 46 8A CE</td>
</tr>
</tbody>
</table>

Note that a value of zero in the R₂ field of any of the RR-format instructions indicates that the branching function is not to be performed; it does not refer to register 0. Likewise, a value of zero in the R₁ field of the BSM instruction indicates that the old value of PSW bit 32 is not to be saved and that register 0 is to be left unchanged. Register 0 can be designated by the R₁ field of instructions BAL, BALR, BAS, BASR, and BASSM, however. In the RX-format branch instructions, branching occurs independent of whether there is a value of zero in the B₂ field or X₂ field of the instruction. However, when the field is zero, instead of using the contents of general register 0, a value of zero is used for that component of address generation.

**Programming Note:** It should be noted that execution of BAL in the 24-bit addressing mode results in bit 0 of register 5 being set to one. This is because the ILC for an RX-format instruction is 10. This is the only case in which bit zero of the return register does not correctly reflect the addressing mode of the caller. Thus, BSM may be used to return for BALR, BAS, BASR, and BASSM in both the 24-bit and the 31-bit addressing modes, but it cannot be used to return if the program was called by using BAL in the 24-bit addressing mode.

### Other BALR and BASR Examples

The BALR or BASR instruction with the R₂ field set to zero may be used to load a register for use as a base register. For example, in the assembler language, the two statements:

- BALR 15,0
- USING *,15

or

- BASR 15,0
- USING *,15
indicate that the address of the next sequential instruction following the BALR or BASR instruction will be placed in register 15, and that the assembler may use register 15 as a base register until otherwise instructed. (The USING statement is an “assembler instruction” and is thus not a part of the object program.)

BRANCH AND STACK (BAKR)

The semiprivileged BRANCH AND STACK instruction facilitates linkage between subroutines by saving status in a linkage-stack state entry (sometimes called a branch state entry to distinguish it from a program-call state entry). When BRANCH AND STACK has been used, the return from the called program is made by means of the PROGRAM RETURN instruction. PROGRAM RETURN restores access registers 2-14, general registers 2-14, and the PSW with values saved in the state entry, except that it leaves the PER mask unchanged and sets the condition code to an unpredictable value. The use of BRANCH AND STACK is discussed in “Branching Using the Linkage Stack” on page 5-72.

BRANCH AND STACK can be used to perform a calling linkage, or it can be used at or near the entry point of the called program, depending on whether the R1 field of the instruction is zero or nonzero, respectively. If the R1 field is zero, bits 32-63 of the PSW saved in the state entry indicate the current addressing mode (24-bit or 31-bit) and the address of the next sequential instruction after the BRANCH AND STACK instruction or an EXECUTE instruction. If the R1 field is nonzero, bits 32-63 of the PSW saved in the state entry are set with a value generated from the contents of general register R2: bit 32 of the PSW is set equal to bit 0 of the register, and bits 1-31 of the PSW are set with an address generated from bits 1-31 of the register under the control of bit 0 of the register. Bits 32-63 of the PSW saved in the state entry are referred to in the following examples as the branch value.

The branch address for the instruction is generated from the contents of general register R2 under the control of the current addressing mode. Bit 0 of general register R2 does not affect the operation. If the R2 field of the instruction is zero, the operation is performed without branching.

In addition to saving a complete PSW (except with an unpredictable PER mask) in the state entry, BRANCH AND STACK saves the new value of bits 32-63 of the current PSW in the state entry. Bits 32-63 are referred to in the following examples as the branch value.

The following examples contain cases in which bit 32 of the current PSW is either zero or one (24-bit or 31-bit addressing) before BRANCH AND STACK is executed and in which bit 0 of the general register designated by a nonzero R1 or R2 field is either zero or one.

BAKR Example 1

This example shows BAKR used in a calling program. BAKR performs a branch, and the return is to be to the next sequential instruction.

The format of the BAKR instruction is:

Machine Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B240</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Assembler Format

BAKR /zerodot,6

Assume four cases of initial values, as follows:

PSW (32-63) Register 6

1. 00 00 10 D6 02 46 8A CE
2. 00 00 10 D6 82 46 8A CE
3. 80 00 10 D6 02 46 8A CE
4. 80 00 10 D6 82 46 8A CE

The results in the four cases are as follows:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Branch Value and PSW (32-63)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 00 00 10 DA</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>2. 00 00 10 DA</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>3. 80 00 10 DA</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>4. 80 00 10 DA</td>
<td>82 46 8A CE</td>
</tr>
</tbody>
</table>
BAKR Example 2
This example shows BAKR used in a called program. BAKR does not perform a branch, and the return is to be as specified in general register R₁.

The format of the BAKR instruction is:

<table>
<thead>
<tr>
<th>Machine Format</th>
<th>Machine Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code R₁ R₂</td>
<td>B240 5 0</td>
</tr>
</tbody>
</table>

Assembler Format

BAKR 5,0

Assume four cases of initial values, as follows:

<table>
<thead>
<tr>
<th>Register 5</th>
<th>PSW (32-63)</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 00 10 D6</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>04 00 10 D6</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>84 00 10 D6</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>84 00 10 D6</td>
<td>82 46 8A CE</td>
</tr>
</tbody>
</table>

The results in the four cases are as follows:

<table>
<thead>
<tr>
<th>Return Value and PSW (32-63)</th>
<th>Branch Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 10 D6 00 46 8A D2</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>00 00 10 D6 82 46 8A D2</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>84 00 10 D6 00 46 8A D2</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>84 00 10 D6 82 46 8A D2</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
</tbody>
</table>

BAKR Example 3
This example shows BAKR used in a called program. BAKR performs a branch, and the return is to be as specified in general register R₁.

The format of the BAKR instruction is:

<table>
<thead>
<tr>
<th>Machine Format</th>
<th>Machine Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code R₁ R₂</td>
<td>B240 5 6</td>
</tr>
</tbody>
</table>

Assembler Format

BAKR 5,6

Assume eight cases of initial values, as follows:

<table>
<thead>
<tr>
<th>Register 5</th>
<th>Register 6</th>
<th>PSW (32-63)</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 00 10 D6</td>
<td>06 99 99 00</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>04 00 10 D6</td>
<td>06 99 99 00</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>04 00 10 D6</td>
<td>86 99 99 00</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>04 00 10 D6</td>
<td>86 99 99 00</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>84 00 10 D6</td>
<td>06 99 99 00</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>84 00 10 D6</td>
<td>06 99 99 00</td>
<td>82 46 8A CE</td>
</tr>
<tr>
<td>84 00 10 D6</td>
<td>86 99 99 00</td>
<td>00 46 8A CE</td>
</tr>
<tr>
<td>84 00 10 D6</td>
<td>86 99 99 00</td>
<td>82 46 8A CE</td>
</tr>
</tbody>
</table>

The results in the eight cases are as follows:

<table>
<thead>
<tr>
<th>Return Value and PSW (32-63)</th>
<th>Branch Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 10 D6 00 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>00 00 10 D6 86 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>00 00 10 D6 00 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>00 00 10 D6 86 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>84 00 10 D6 00 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>84 00 10 D6 86 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>84 00 10 D6 00 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
<tr>
<td>84 00 10 D6 86 99 99 00</td>
<td>84 00 10 D6 86 99 99 00</td>
</tr>
</tbody>
</table>

BRANCH ON CONDITION (BC, BCR)

The BRANCH ON CONDITION instruction tests the condition code to see whether a branch should or should not occur. The branch occurs only if the current condition code corresponds to a one bit in a mask specified by the instruction.

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Instruction (Mask) Bit</th>
<th>Mask Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, assume that an ADD (A or AR) operation has been performed and that a branch to address 6050 is desired if the sum is zero or less (condition code is 0 or 1). Also assume:

Register 10 contains 00 00 50 00.
Register 11 contains 00 00 10 00.

The RX form of the instruction performs the required test (and branch if necessary) when written as:
A mask of 12 means that there are ones in instruction bits 8 and 9 and zeros in bits 10 and 11, so that branching takes place when the condition code is either 0 or 1.

A mask of 15 would indicate a branch on any condition (an unconditional branch). A mask of zero would indicate that no branch is to occur (a no-operation).

(See also "Linkage Instructions (BAL, BALR, BAS, BASR, BASSM, BSM)" on page A-8 for an example of the BCR instruction.)

BRANCH ON COUNT (BCT, BCTR)

The BRANCH ON COUNT instruction is often used to execute a program loop for a specified number of times. For example, assume that the following represents some lines of coding in an assembler-language program:

```
... 
LUPE AR 8,1 
... 
BACK BCT 6,LUPE
```

where register 6 contains 00 00 00 03 and the address of LUPE is 6826. Assume that, in order to address this location, register 10 is used as a base register and contains 00 00 68 00.

The format of the BCT instruction is:

<table>
<thead>
<tr>
<th>Machine Format</th>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCT 6,X'50'(11,10)</td>
<td>46</td>
<td>6</td>
<td>0</td>
<td>A</td>
<td>026</td>
<td></td>
</tr>
</tbody>
</table>

BRANCH ON INDEX HIGH (BXH)

BXH Example 1

The BRANCH ON INDEX HIGH instruction is an index-incrementing and loop-controlling instruction that causes a branch whenever the sum of an index value and an increment value is greater than some compare value. For example, assume that:

- Register 4 contains 00 00 00 8A = 138 = the index.
- Register 6 contains 00 00 00 02 = 2 = the increment.
- Register 7 contains 00 00 00 AA = 170 = the compare value.
- Register 10 contains 00 00 71 30 = the branch address.

The format of the BXH instruction is:

<table>
<thead>
<tr>
<th>Machine Format</th>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BXH 4,6,/zerodot(1/zerodot)</td>
<td>86</td>
<td>4</td>
<td>6</td>
<td>A</td>
<td>000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When the instruction is executed, first the contents of register 6 are added to register 4, second the sum is compared with the contents of register 7, and third the decision whether to branch is made. After execution:

- Register 4 contains 00 00 00 8C = 140.
- Registers 6 and 7 are unchanged.

Since the new value in register 4 is not yet greater than the value in register 7, the branch to address 7130 is not taken. Repeated use of the instruction...
will eventually cause the branch to be taken when the value in register 4 reaches 172₁₀.

**BXH Example 2**

When the register used to contain the increment is odd, that register also becomes the compare-value register. The following assembler-language subroutine illustrates how this may be used to search a table.

<table>
<thead>
<tr>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>2 Bytes</td>
</tr>
<tr>
<td>ARG1</td>
</tr>
<tr>
<td>ARG2</td>
</tr>
<tr>
<td>ARG3</td>
</tr>
<tr>
<td>ARG4</td>
</tr>
<tr>
<td>ARG5</td>
</tr>
<tr>
<td>ARG6</td>
</tr>
</tbody>
</table>

Assume that:

Register 8 contains the search argument.

Register 9 contains the width of the table in bytes (00 00 00 04).

Register 10 contains the length of the table in bytes (00 00 00 18).

Register 11 contains the starting address of the table.

Register 14 contains the return address to the main program.

As the following subroutine is executed, the argument in register 8 is successively compared with the arguments in the table, starting with argument 6 and working backward to argument 1. If an equality is found, the corresponding function replaces the argument in register 8. If an equality is not found, zero replaces the argument in register 8.

```
SEARCH LNR 9,9
NOTEQUAL BXH 10,9,LOOP
NOTFOUND SR 8,8
BCR 15,14
LOOP CH 8,0(10,11)
   BC 7,NOTEQUAL
   LH 8,2(10,11)
   BCR 15,14
```

The first instruction (LNR) causes the value in register 9 to be made negative. After execution of this instruction, register 9 contains FF FF FF FC = -4₁₀. Considering the case when no equality is found, the BXH instruction will be executed seven times. Each time BXH is executed, a value of -4 is added to register 10, thus reducing the value in register 10 by 4. The new value in register 10 is compared with the -4 value in register 9. The branch is taken each time until the value in register 10 is -4. Then the branch is not taken, and the SR instruction sets register 8 to zero.

**BRANCH ON INDEX LOW OR EQUAL (BXLE)**

The BRANCH ON INDEX LOW OR EQUAL instruction performs the same operation as BRANCH ON INDEX HIGH, except that branching occurs when the sum is lower than or equal to (instead of higher than) the compare value. As the instruction which increments and tests an index value in a program loop, BXLE is useful at the end of the loop and BXH at the beginning. The following assembler-language routines illustrate loops with BXLE.

**BXLE Example 1**

Assume that a group of ten 32-bit signed binary integers are stored at consecutive locations, starting at location GROUP. The integers are to be added together, and the sum is to be stored at location SUM.

```
SR 5,5 Set sum to zero
LA 6,GROUP Load first address
SR 7,7 Set index to zero
LA 8,4 Load increment 4
LA 9,39 Load compare value
LOOP A 5,/zerodot(7,6) Add integer to sum
   BXLE 7,8,LOOP Test end of loop
ST 5,SUM Store sum
```

The two-instruction loop contains an ADD (A) instruction which adds each integer to the contents of general register 5. The ADD instruction uses the contents of general register 7 as an index value to modify the starting address obtained from register 6. Next, BXLE increments the index value by 4, the increment previously loaded into register 8, and compares it with the compare value in register 9, the odd register of this even-odd pair. The compare value was previously set to 39, which is one less than the number of bytes in the data area; this is also the address, relative to the starting address, of the rightmost byte of the last integer to be added. When the last integer has been added, BXLE increments the index value to the next relative address (40),
which is found to be greater than the compare value (39) so that no branching takes place.

**BXLE Example 2**
The technique illustrated in Example 1 is restricted to loops containing instructions in the RX instruction format. That format allows both a base register and an index register to be specified (double indexing).

For instructions in other formats, where an index register cannot be specified, the previous technique may be modified by having the address itself serve as the index value in a BXLE instruction and by using as the compare value the address of the last byte rather than its relative address. The base register then provides the address directly at each iteration of the loop, and it is not necessary to specify a second register to hold the index value (single indexing).

In the following example, an AND (NI) instruction in the SI instruction format sets to zero the rightmost bit of each of the same group of integers as in Example 1, thus making all of them even. The $I_2$ field of the NI instruction contains the byte X'FE', which consists of seven ones and a zero. That byte is ANDed into byte 3, the rightmost byte, of each of the integers in turn.

```
LA 6, GROUP Load first address
LA 8, 4 Load increment 4
LA 9, GROUP+39 Load compare value
LOOP NI 3(6), X'FE' AND immediate
BXLE 6, 8, LOOP Test end of loop
```

The technique shown in Example 2 does not work, however, on an ESA/370 system when it is in the 31-bit addressing mode and the data is located at the rightmost end of a 31-bit address space. In this case, the compare value would be set to $2^{31}-1$, which is the largest possible 32-bit signed binary value. The reason the technique does not work is that the BXLE and BXH instructions treat their operands as 32-bit signed binary integers. When the address in general register 6 reaches the value $2^{31}-4$, BXLE increments it to a value that is interpreted as -$2^{31}$, rather than $2^{31}$, and the comparison remains low, which causes looping to continue indefinitely.

This situation can be avoided by not allowing data areas to extend to the rightmost location in a 31-bit address space or by using other techniques; these may include double indexing when possible, as in Example 1, or starting at the end and stepping downward through the data area with a negative increment.

**COMPARE AND FORM CODEWORD (CFC)**

See “Sorting Instructions” on page A-51.

**COMPAREHALFWORD (CH)**
The COMPARE HALFWORD instruction compares a 16-bit signed binary integer in storage with the contents of a register. For example, assume that:

- Register 4 contains FF FF 80 00 = -32,768.
- Register 13 contains 00 01 60 50.
- Storage locations 16080-16081 contain 8000 = -32,768.

When the instruction:

```
CH 4, X'30'(0,13)
```

is executed, the contents of locations 16080-16081 are fetched, expanded to 32 bits (the sign bit is propagated to the left), and compared with the contents of register 4. Because the two numbers are equal, condition code 0 is set.

**COMPARE LOGICAL (CL, CLC, CLI, CLR)**
The COMPARE LOGICAL instruction differs from the signed-binary comparison instructions (C, CH, CR) in that all quantities are handled as unsigned binary integers or as unstructured data.

**CLC Example**
The COMPARE LOGICAL (CLC) instruction can be used to perform the byte-by-byte comparison of storage fields up to 256 bytes in length. For example, assume that the following two fields of data are in storage:
Field 1
1886 1891
├──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
│D1│D6│C8│D5│E2│D6│D5│6B│C1│4B│C2│4B│
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘

Field 2
1900 190B
├──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
│D1│D6│C8│D5│E2│D6│D5│6B│C1│4B│C3│4B│
└──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘

Also assume:
- Register 9 contains 00 00 18 80.
- Register 7 contains 00 00 19 00.

Execution of the instruction:

Machine Format
Op Code L B/basesuperone D/basesuperone B/basesupertwo D/basesupertwo
├────────┬────┬────┬────┬────┬────┐
│ D5 │ /zerodotB │ 9 │ /zerodot/zerodot6│ 7 │ /zerodot/zerodot/zerodot│
└────────┴────┴────┴────┴────┴────┘

Assembler Format
Op Code D/basesuperone(L,B/basesuperone),D/basesupertwo(B/basesupertwo)
------------------------------------------------------------
CLC 6(12,9),/zerodot(7)
sets condition code 1, indicating that the first operand (the quantity in main storage) is lower than the second (immediate) operand.

CLR Example
Assume that:
- Register 4 contains 00 00 00 01 = 1.
- Register 7 contains FF FF FF FF = 2^{32} - 1.

Execution of the instruction:

Machine Format
Op Code R/basesuperone R/basesupertwo
├────────┬────┬────┤
│ 15 │ 4 │ 7 │
└────────┴────┴────┘

Assembler Format
Op Code R/basesuperone,R/basesupertwo
---------------------------
CLR 4,7
sets condition code 1. Condition code 1 indicates that the first operand is lower than the second.

If, instead, the signed-binary comparison instruction COMPARE (CR) had been executed, the contents of register 4 would have been interpreted as +1 and the contents of register 7 as -1. Thus, the first operand would have been higher, so that condition code 2 would have been set.

COMPARE LOGICAL CHARACTERS UNDER MASK (CLM)

The COMPARE LOGICAL CHARACTERS UNDER MASK (CLM) instruction provides a means of comparing bytes selected from a general register to a contiguous field of bytes in storage. The M_3 field of the CLM instruction is a
four-bit mask that selects zero to four bytes from a general register, each mask bit corresponding, left to right, to a register byte. In the comparison, the register bytes corresponding to ones in the mask are treated as a contiguous field. The operation proceeds left to right. For example, assume that:

Storage locations 10200-10202 contain F0 BC 7B.

Register 12 contains 00 01 00 00.

Register 6 contains F0 BC 5C 7B.

Execution of the instruction:

Machine Format
Op Code   R1  M3  B2  D2
BD        6   D   C  200

Assembler Format
Op Code   R1,M3,D2(B2)
CLM  6,B'1101',X'200'(12)

causes the following comparison:

Register 6: F0 BC 5C 7B
Mask M3:  1 1 0 1
          -- -- --

F0 BC 7B

Storage locations 10200-10202: F0 BC 7B

Because the selected bytes are equal, condition code 0 is set.

**COMPARE LOGICAL LONG (CLCL)**

The COMPARE LOGICAL LONG instruction is used to compare two operands in storage, byte by byte. Each operand can be of any length. Two even-odd pairs of general registers (four registers in all) are used to locate the operands and to control the execution of the CLCL instruction, as illustrated in the following diagram. The first register of each pair must be an even register, and it contains the storage address of an operand. The odd register of each pair contains the length of the operand it covers, and the leftmost byte of the second-operand odd register contains a padding byte which is used to extend the shorter operand, if any, to the same length as the longer operand.

The following illustrates the assignment of registers in the 24-bit addressing mode:

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>First-Operand Address</td>
</tr>
<tr>
<td>R1+1</td>
<td>First-Operand Length</td>
</tr>
<tr>
<td>R2</td>
<td>Second-Operand Address</td>
</tr>
<tr>
<td>R2+1</td>
<td>Second-Operand Length</td>
</tr>
</tbody>
</table>
```

In the 31-bit addressing mode, the operand addresses would be in bit positions 1-31 of the even registers shown above.

Since the CLCL instruction may be interrupted during execution, the interrupting program must preserve the contents of the four registers for use when the instruction is resumed.

The following instructions set up two register pairs to control a text-string comparison. For example, assume:

```
Operand 1
Address: 2080016
Length: 10010

Operand 2
Address: 20A0016
Length: 13210

Padding Byte
Address: 2000316
Length: 1
Value: 4016
```

Register 12 contains 00 02 00 00.

The setup instructions are:
Register pair 4,5 defines the first operand. Bits 8-31 of register 4 contain the storage address of the start of an EBCDIC text string, and bits 8-31 of register 5 contain the length of the string, in this case 100 bytes.

Register pair 8,9 defines the second operand, with bits 8-31 of register 8 containing the starting location of the second operand and bits 8-31 of register 9 containing the length of the second operand, in this case 132 bytes. Bits 0-7 of register 9 contain an EBCDIC blank character (X'40') to pad the shorter operand. In this example, the padding byte is used in the first operand, after the 100th byte, to compare with the remaining bytes in the second operand.

With the register pairs thus set up, the format of the CLCL instruction is:

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

**Assembler Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁, R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLCL</td>
<td>4,8</td>
</tr>
</tbody>
</table>

When this instruction is executed, the comparison starts at the left end of each operand and proceeds to the right. The operation ends as soon as an inequality is detected or the end of the longer operand is reached.

If this CLCL instruction is interrupted after 60 bytes have compared equal, the operand lengths in registers 5 and 9 will have been decremented to 40 and 72, respectively. The operand addresses in registers 4 and 8 will have been incremented to X'2083C' and X'20A3C'; the leftmost byte of registers 4 and 8 will have been set to zero. The padding byte X'40' remains in register 9. When the CLCL instruction is reexecuted with these register contents, the comparison resumes at the point of interruption.

Now, assume that the instruction is interrupted after 110 bytes. That is, the first 100 bytes of the second operand have compared equal to the first operand, and the next 10 bytes of the second operand have compared equal to the padding byte (blank). The residual operand lengths in registers 5 and 9 are 0 and 22, respectively, and the operand addresses in registers 4 and 8 are X'20864' (the value when the first operand was exhausted) and X'20A6E' (the current value for the second operand).

When the comparison ends, the condition code is set to 0, 1, or 2, depending on whether the first operand is equal to, less than, or greater than the second operand, respectively.

When the operands are unequal, the addresses in registers 4 and 8 indicate the bytes that caused the mismatch.

**COMPARE LOGICAL STRING (CLST)**

The COMPARE LOGICAL STRING instruction is used to compare a first operand designated by general register R₁ and a second operand designated by general register R₂. The comparison is made left to right, byte by byte, until unequal bytes are compared, an ending character specified in general register 0 is encountered in either operand, or a CPU-determined number of bytes have been compared. The condition code is set to 0 if the two operands are equal, to 1 if the first operand is low, to 2 if the second operand is low, or to 3 if a CPU-determined number of bytes have been compared. If the ending character is found in both operands simultaneously, the operands are equal. If it is found in only one operand, that operand is low.

When condition code 1 or 2 is set, the addresses of the last bytes processed in the first and second operands are placed in general registers R₁ and R₂, respectively. These are the addresses of unequal bytes in the two operands, or they are the
address of an ending character in one operand
and of the byte in the corresponding byte position
in the other operand. When condition code 3 is
set, the addresses of the next bytes to be proc-
essed are placed in the registers. When condition
code 0 is set, the contents of the registers remain
unchanged.

Following are examples of first and second oper-
ands beginning at decimal locations 1000 and
2000, respectively. The addresses in general reg-
isters R\textsubscript{1} and R\textsubscript{2} are 1000 and 2000, respectively.
The ending character in general register 0 is 00
hex (as in the C programming language). The
values of the operand bytes are shown in hex,
and the resulting condition code and final contents
of general registers R\textsubscript{1} and R\textsubscript{2} are shown.

**Example 1**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

CC: 0; (R\textsubscript{1}): 1000; (R\textsubscript{2}): 2000

**Example 2**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

CC: 1; (R\textsubscript{1}): 1003; (R\textsubscript{2}): 2003

**Example 3**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

CC: 2; (R\textsubscript{1}): 1003; (R\textsubscript{2}): 2003

**Example 4**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

CC: 3; (R\textsubscript{1}): 1256; (R\textsubscript{2}): 2256

**Example 5**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

CC: 1; (R\textsubscript{1}): 1000; (R\textsubscript{2}): 2000

**Example 6**

Assuming that the CPU-determined number of
bytes compared is 256:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

CC: 2; (R\textsubscript{1}): 1000; (R\textsubscript{2}): 2000

**Example 7**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

CC: 0; (R\textsubscript{1}): 1000; (R\textsubscript{2}): 2000

CONVERT TO BINARY (CVB)

The CONVERT TO BINARY instruction converts
an eight-byte, packed-decimal number into a
signed binary integer and loads the result into a
general register. After the conversion operation is
completed, the number is in the proper form for
use as an operand in signed binary arithmetic.
For example, assume:

Storage locations 7608-760F contain a
decimal number in the packed format: 00 00
00 00 00 25 59 4C (+25,594).

The contents of register 7 are not significant.
Register 13 contains 00 00 76 00.

The format of the conversion instruction is:

**Machine Format**

| 4F | 7 | 0 | D | 008 |

**Assembler Format**

| 4F | 7 |

CVB 7,8(0,13)

After the instruction is executed, register 7 con-
tains 00 00 63 FA.

CONVERT TO DECIMAL (CVD)

The CONVERT TO DECIMAL instruction is the
opposite of the CONVERT TO BINARY instruc-
tion. CVD converts a signed binary integer in a
register to packed decimal and stores the eight-
byte result. For example, assume:

Register 1 contains the signed binary integer:
00 00 0F 0F.

Register 13 contains 00 00 76 00.
The format of the instruction is:

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>4E</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>008</td>
</tr>
</tbody>
</table>

**Assembler Format**

Op Code R₁,D₂(X₂,B₂)

CVD 1,8(/zerodot,13)

After the instruction is executed, storage locations 7608-760F contain 00 00 00 00 03 85 5C (+3855).

The plus sign generated is the preferred plus sign, 1100₂.

**DIVIDE (D, DR)**

The DIVIDE instruction divides the dividend in an even-odd register pair by the divisor in a register or in storage. Since the instruction assumes the dividend to be 64 bits long, it is important first to extend a 32-bit dividend on the left with bits equal to the sign bit. For example, assume that:

- Storage locations 3550-3553 contain 00 00 08 DE = 2270₁₀ (the dividend).
- Storage locations 3554-3557 contain 00 00 32 = 50₁₀ (the divisor).

The initial contents of registers 6 and 7 are not significant.

Register 8 contains 00 00 35 50.

The following assembler-language statements load the registers properly and perform the divide operation:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>L 6,0(0,8)</td>
<td>Places 00 00 08 DE into register 6.</td>
</tr>
<tr>
<td>SRDA 6,32(0)</td>
<td>Shifts 00 00 08 DE into register 7. Register 6 is filled with zeros (sign bits).</td>
</tr>
<tr>
<td>D 6,4(0,8)</td>
<td>Performs the division.</td>
</tr>
</tbody>
</table>

The machine format of the above DIVIDE instruction is:

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>5D</td>
<td>6</td>
<td>0</td>
<td>8</td>
<td>004</td>
</tr>
</tbody>
</table>

After the instructions listed above are executed:

- Register 6 contains 00 00 00 14 = 20₁₀ = the remainder.
- Register 7 contains 00 00 00 2D = 45₁₀ = the quotient.

Note that if the dividend had not been first placed in register 6 and shifted into register 7, register 6 might not have been filled with the proper dividend-sign bits (zeros in this example), and the DIVIDE instruction might not have given the expected results.

**EXCLUSIVE OR (X, XC, XI, XR)**

When the Boolean operator EXCLUSIVE OR is applied to two bits, the result is one when either, but not both, of the two bits is one; otherwise, the result is zero. When two bytes are EXCLUSIVE ORed, each pair of bits is handled separately; there is no connection from one bit position to another. The following is an example of the EXCLUSIVE OR of two bytes:

First-operand byte: /zerodot/zerodot 11 /zerodot1/zerodot1/basesupertwo

Second-operand byte: /zerodot1/zerodot1 11/zerodot/zerodot/basesupertwo

Result byte: /zerodot11/zerodot 1/zerodot/zerodot1/basesupertwo

**XC Example**

The EXCLUSIVE OR (XC) instruction can be used to exchange the contents of two areas in storage without the use of an intermediate storage area. For example, assume two three-byte fields in storage:

Field 1 00 17 90

Field 2 00 14 01

Execution of the instruction (assume that register 7 contains 00 00 03 58):
Machine Format

Op Code  L   B₁   D₁   B₂   D₂
----------  ----  ----  ----  ----  ----
  D7 02  7  001  7  008

Assembler Format

Op Code  D₁(L,B₁),D₂(B₂)
----------  -------------------------------
          XC  1(3,7),8(7)

Field 1 is EXCLUSIVE ORed with field 2 as follows:

Field 1: 00000000 00010111 10010000₁₂ = 00 17 90₁₆
Field 2: 00000000 00010100 0000000₁₂ = 00 14 01₁₆

Result: 00000000 0000001₁₁ 10010000₁₂ = 00 03 9₁₁₆

The result replaces the former contents of field 1. Field 1 now contains the original value of field 2. Condition code 1 is set to indicate a nonzero result.

Now, execution of the instruction:

Machine Format

Op Code  L   B₁   D₁   B₂   D₂
----------  ----  ----  ----  ----  ----
  D7 02  7  008  7  001

Assembler Format

Op Code  D₁(L,B₁),D₂(B₂)
----------  -------------------------------
          XC  8(3,7),1(7)

produces the following result:

Field 1: 00000000 00010111 10010000₁₂ = 00 17 90₁₆
Field 2: 00000000 00010100 0000000₁₂ = 00 14 01₁₆

Result: 00000000 0000001₁₁ 10010000₁₂ = 00 03 9₁₁₆

The result of this operation replaces the former contents of field 1. Field 1 now contains the original value of field 2. Condition code 1 is set to indicate a nonzero result.

Lastly, execution of the instruction:

Machine Format

Op Code  L   B₁   D₁   B₂   D₂
----------  ----  ----  ----  ----  ----
  D7 02  7  001  7  008

Assembler Format

Op Code  D₁(L,B₁),D₂(B₂)
----------  -------------------------------
          XI  2(9),X'81'

When the instruction is executed, the byte in storage is EXCLUSIVE ORed with the immediate byte (the i₂ field of the instruction):

Location 8082: 0110 100₁₂
Immediate byte: 1000 000₁₂

Result: 1110 100₀₀₂

The resulting byte is stored back in location 8082. Condition code 1 is set to indicate a nonzero result.

Notes:

1. With the XC instruction, fields up to 256 bytes in length can be exchanged.
2. With the XR instruction, the contents of two registers can be exchanged.
3. Because the X instruction operates storage to
register only, an exchange cannot be made solely by the use of X.

4. A field EXCLUSIVE ORed with itself is cleared to zeros.

5. For additional examples of the use of EXCLUSIVE OR, see "Hexadecimal-Floating-Point-Number Conversion" on page A-42.

EXECUTE (EX)

The EXECUTE instruction causes one target instruction in main storage to be executed out of sequence without actually branching to the target instruction. Unless the R₁ field of the EXECUTE instruction is zero, bits 8-15 of the target instruction are ORed with bits 24-31 of the R₁ register before the target instruction is executed. Thus, EXECUTE may be used to supply the length field for an SS instruction without modifying the SS instruction in storage. For example, assume that a MOVE (MVC) instruction is the target that is located at address 3820, with a format as follows:

Machine Format

\[
\begin{array}{cccccc}
D2 & 00 & C & 003 & D & 000 \\
\end{array}
\]

Assembler Format

\[
\text{MVC } 3(4,12),/\text{zerodot}/(13) 
\]

where register 12 contains 00 00 89 13 and register 13 contains 00 00 90 A0.

Further assume that at storage address 5000, the following EXECUTE instruction is located:

Machine Format

\[
\begin{array}{cccccc}
44 & 1 & 0 & A & 000 \\
\end{array}
\]

Assembler Format

\[
\text{EX } 1,0(0,10) 
\]

where register 10 contains 00 00 38 20 and register 1 contains 00 0F 00 03.

When the instruction at 5000 is executed, the rightmost byte of register 1 is ORed with the second byte of the target instruction:

Instruction byte: 0000 0000₂ = 00
Register byte: 0000 0011₂ = 03

Result: 0000 0011₂ = 03

causing the instruction at 3820 to be executed as if it originally were:

Machine Format

\[
\begin{array}{cccccc}
D2 & 03 & C & 003 & D & 000 \\
\end{array}
\]

Assembler Format

\[
\text{MVC } 3(4,12),/\text{zerodot}/(13) 
\]

However, after execution:

Register 1 is unchanged.
The instruction at 3820 is unchanged.
The contents of the four bytes starting at location 90A0 have been moved to the four bytes starting at location 8916.
The CPU next executes the instruction at address 5004 (PSW bits 40-63 contain 00 50 04).

INSERT CHARACTERS UNDER MASK (ICM)

The INSERT CHARACTERS UNDER MASK (ICM) instruction may be used to replace all or selected bytes in a general register with bytes from storage and to set the condition code to indicate the value of the inserted field.

For example, if it is desired to insert a three-byte address from FIELDA into register 5 and leave the leftmost byte of the register unchanged, assume:

Machine Format

\[
\begin{array}{cccccc}
BF & 5 & 7 & \ast & \ast & \ast \\
\end{array}
\]

where register 10 contains 00 00 38 20 and register 1 contains 00 0F 00 03.
The contents of register 10 are not significant.

Storage locations 21003-21006 contain 00 00 AB CD.

To load register 10, the RX form of the instruction can be used:

<table>
<thead>
<tr>
<th>Machine Format</th>
<th>Op Code</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>58</td>
<td>A</td>
<td>5</td>
<td>6</td>
<td>000</td>
</tr>
</tbody>
</table>

After the instruction is executed, register 10 contains 00 00 AB CD.

**LOAD ADDRESS (LA)**

The LOAD ADDRESS instruction provides a convenient way to place a nonnegative binary integer up to $4095_{10}$ in a register without first defining a constant and then using it as an operand. For example, the following instruction places the number $2048_{10}$ in register 1:

<table>
<thead>
<tr>
<th>Machine Format</th>
<th>Op Code</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>41</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>800</td>
</tr>
</tbody>
</table>

The LOAD ADDRESS instruction can also be used to increment a register by an amount up to $4095_{10}$ specified in the D2 field. Depending on the addressing mode, only the rightmost 24 or 31 bits of the sum are retained, however. The leftmost bits of the 32-bit result are set to zeros. For example, assume that register 5 contains 00 12 34 56.

The instruction:
Machine Format
Op Code   R1  X2  B2  D2
41  5  0  5  00A

Assembler Format
Op Code   R1,D2(X2,B2)
LA  5,10(0,5)

adds 10 (decimal) to the contents of register 5 as follows:
Register 5 (old): 00 12 34 56
D2 field: 00 00 00 0A
Register 5 (new): 00 12 34 60

The register may be specified as either B2 or X2. Thus, the instruction LA 5,10(5,0) produces the same result.

As the most general example, the instruction LA 6,10(5,4) forms the sum of three values: the contents of register 4, the contents of register 5, and a displacement of 10 and places the 24-bit or 31-bit sum with zeros appended on the left in register 6.

LOAD HALFWORD (LH)

The LOAD HALFWORD instruction places unchanged a halfword from storage into the right half of a register. The left half of the register is loaded with zeros or ones according to the sign (leftmost bit) of the halfword.

For example, assume that the two bytes in storage locations 1803-1804 are to be loaded into register 6. Also assume:
The contents of register 6 are not significant.
Register 14 contains 00 00 18 03.
Locations 1803-1804 contain 00 20.
The instruction required to load the register is:
Machine Format
Op Code   R1  X2  B2  D2
48  6  0  E  000

Assembler Format
Op Code   R1,D2(X2,B2)
LH  6,0(0,14)

After the instruction is executed, register 6 contains 00 00 00 20. If locations 1803-1804 had contained a negative number, for example, A7 B6, a minus sign would have been propagated to the left, giving FF FF A7 B6 as the final result in register 6.

MOVE (MVC, MVI)

MVC Example

The MOVE (MVC) instruction can be used to move data from one storage location to another. For example, assume that the following two fields are in storage:
Field 1
<table>
<thead>
<tr>
<th>2</th>
<th>48</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
<td>C3</td>
</tr>
</tbody>
</table>
Field 2
<table>
<thead>
<tr>
<th>2</th>
<th>40</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F2</td>
<td>F3</td>
</tr>
</tbody>
</table>

Also assume:
Register 1 contains 00 00 20 48.
Register 2 contains 00 00 38 40.

With the following instruction, the first eight bytes of field 2 replace the first eight bytes of field 1:
Machine Format
Op Code   L  B1  D1  B2  D2
D2  07  1  000  2  000

Assembler Format
Op Code   D1(L,B1),D2(B2)
MVC  0(8,1),0(2)

After the instruction is executed, field 1 becomes:
Field 1
<table>
<thead>
<tr>
<th>2</th>
<th>48</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F2</td>
<td>F3</td>
</tr>
</tbody>
</table>
Field 2 is unchanged.

MVC can also be used to propagate a byte through a field by starting the first-operand field one byte location to the right of the second-operand field. For example, suppose that an area in storage starting with address 358 contains the following data:
With the following MVC instruction, the zeros in location 358 can be propagated throughout the entire field (assume that register 11 contains 00 00 03 58):

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D2</td>
<td>07</td>
<td>B</td>
<td>001</td>
<td>B</td>
</tr>
</tbody>
</table>

**Assembler Format**

Op Code D1(L,B1),D2(B2)

MVC 1(8,11),0(11)

Because MVC is executed as if one byte were processed at a time, the above instruction, in effect, takes the byte at address 358 and stores it at 359 (359 now contains 00), takes the byte at 359 and stores it at 35A, and so on, until the entire field is filled with zeros. Note that an MVI instruction could have been used originally to place the byte of zeros in location 358.

**Notes:**

1. Although the field occupying locations 358-360 contains nine bytes, the length coded in the assembler format is equal to the number of moves (one less than the field length).
2. The order of operands is important even though only one field is involved.

**MVI Example**

The MOVE (MVI) instruction places one byte of information from the instruction stream into storage. For example, the instruction:

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>I2</th>
<th>B1</th>
<th>D1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>92</td>
<td>58</td>
<td>1</td>
</tr>
</tbody>
</table>

**Assembler Format**

Op Code D1(B1),I2

MVI 0(1),C'$'

MVC may be used, in conjunction with the instruction EDIT AND MARK, to insert the EBCDIC code for a dollar symbol at the storage address contained in general register 1 (see also the example for EDIT AND MARK).

**MOVE INVERSE (MVCIN)**

The MOVE INVERSE (MVCIN) instruction can be used to move data from one storage location to another while reversing the order of the bytes within the field. For example, assume that the following two fields are in storage:

Field 1

<table>
<thead>
<tr>
<th>2048</th>
<th>2052</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
<td>C4</td>
</tr>
<tr>
<td>C5</td>
<td>C6</td>
</tr>
<tr>
<td>C7</td>
<td>C8</td>
</tr>
<tr>
<td>C9</td>
<td>CA</td>
</tr>
<tr>
<td>CA</td>
<td>CB</td>
</tr>
</tbody>
</table>

Field 2

<table>
<thead>
<tr>
<th>3840</th>
<th>3848</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F2</td>
</tr>
<tr>
<td>F3</td>
<td>F4</td>
</tr>
<tr>
<td>F5</td>
<td>F6</td>
</tr>
<tr>
<td>F7</td>
<td>F8</td>
</tr>
<tr>
<td>F9</td>
<td></td>
</tr>
</tbody>
</table>

Also assume:

- Register 1 contains 00 00 20 48.
- Register 2 contains 00 00 38 40.

With the following instruction, the first eight bytes of field 2 replace the first eight bytes of field 1:

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E8</td>
<td>07</td>
<td>1</td>
<td>000</td>
<td>2</td>
</tr>
</tbody>
</table>

**Assembler Format**

Op Code D1(L,B1),D2(B2)

MVCIN 0(8,1),7(2)

After the instruction is executed, field 1 becomes:

Field 1

<table>
<thead>
<tr>
<th>2048</th>
<th>2052</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>F7</td>
</tr>
<tr>
<td>F6</td>
<td>F5</td>
</tr>
<tr>
<td>F4</td>
<td>F3</td>
</tr>
<tr>
<td>F2</td>
<td>F1</td>
</tr>
<tr>
<td>C9</td>
<td>CA</td>
</tr>
<tr>
<td>CA</td>
<td>CB</td>
</tr>
</tbody>
</table>

Field 2 is unchanged.

**Note:** This example uses the same general registers, storage locations, and original values as the first example for MVC. For MVCIN, the second-operand address must designate the rightmost byte of the field to be moved, in this case location 3847. This is accomplished by means of the 7 in the D2 field of the instruction.
MOVE LONG (MVCL)

The MOVE LONG (MVCL) instruction can be used for moving data in storage as in the first example of the MVC instruction, provided that the two operands do not overlap. MVCL differs from MVC in that the address and length of each operand are specified in an even-odd pair of general registers. Consequently, MVCL can be used to move more than 256 bytes of data with one instruction. As an example, assume:

- Register 2 contains 00 0A 00 00.
- Register 3 contains 00 00 08 00.
- Register 8 contains 00 06 00 00.
- Register 9 contains 00 00 08 00.

Execution of the instruction:

Machine Format
Op Code R/basesuperone R/basesupertwo
 ┌────────┬────┬────┐
 │ /zerodotE │ 8 │ 2 │
 └────────┴────┴────┘

Assembler Format
Op Code R/basesuperone,R/basesupertwo
MVCL 8,2

moves 2,048 bytes from locations A0000-A07FF to locations 60000-607FF. Assuming that the CPU is in the 24-bit addressing mode, bits 8-31 of registers 2 and 8 are incremented by 800, and bits 0-7 of registers 2 and 8 are set to zeros. Bits 8-31 of registers 3 and 9 are decremented to zero. Condition code 0 is set to indicate that the operand lengths are equal.

Condition code 2 would have been set to indicate that the first operand was longer than the second.

The technique for setting a field to zeros that is illustrated in the second example of MVC cannot be used with MVCL. If the registers were set up to attempt such an operation with MVCL, no data movement would take place and condition code 3 would indicate destructive overlap.

Instead, MVCL may be used to clear a storage area to zeros as follows. Assume register 8 and 9 are set up as before. Register 3 contains only zeros, specifying zero length for the second operand and a zero padding byte. Register 2 is not used to access storage, and its contents are not significant. Executing the instruction MVCL 8,2 causes locations 60000-607FF to be filled with zeros. Bits 8-31 of register 8 are incremented by 800, and bits 0-7 of registers 2 and 8 are set to zeros. Bits 8-31 of register 9 are decremented to zero, and condition code 2 is set to indicate that the first operand is longer than the second.

MOVE NUMERICS (MVN)

Two related instructions, MOVE NUMERICS and MOVE ZONES, may be used with decimal data in the zoned format to operate separately on the rightmost four bits (the numeric bits) and the leftmost four bits (the zone bits) of each byte. Both are similar to MOVE (MVC), except that MOVE NUMERICS moves only the numeric bits and MOVE ZONES moves only the zone bits.

To illustrate the operation of the MOVE NUMERICS instruction, assume that the following two fields are in storage:

Field A: C6 C7 C8 C9
Field B: F0 F1 F2 F3 F4 F5

Also assume:
- Register 14 contains 00 00 70 90.
- Register 15 contains 00 00 70 40.

After the instruction:
Machine Format
Op Code L B1 D1 B2 D2
D1 03 F 001 E 000

Assembler Format
Op Code D1(L,B1),D2(B2)
MVN 1(4,15),0(14)

is executed, field B becomes:
7041 7046
F6 F7 F8 F9 F4 F5

The numeric bits of the bytes at locations 7090-7093 have been stored in the numeric bits of the bytes at locations 7041-7044. The contents of locations 7090-7093 and 7045-7046 are unchanged.

**MOVE STRING (MVST)**

The MOVE STRING instruction is used to move a second operand designated by general register R2 to a first-operand location designated by general register R1. The movement is made left to right until an ending character specified in general register 0 has been moved or a CPU-determined number of bytes have been moved. The condition code is set to 1 if the ending character was moved or to 3 if a CPU-determined number of bytes were moved.

When condition code 1 is set, the address of the ending character in the first operand is placed in general register R1, and the contents of general register R2 remain unchanged. When condition code 3 is set, the address of the next byte to be processed in the first and second operands is placed in general registers R1 and R2, respectively.

Following is an example program that sets string A equal to the concatenation of string B followed by string C, where the length of each of strings B and C is unknown, and the end of each of strings B and C is indicated by an ending character of 00 hex (as in the C programming language). The program is not written for execution in the access-register mode.

```assembly
L 4,STRAADR
L 5,STRBADR
SR 0,0
LOOP1 MVST 4,5
BC 1,LOOP1
L 5,STRCADR
LOOP2 MVST 4,5
BC 1,LOOP2
[Any instruction]
```

**MOVE WITH OFFSET (MVO)**

MOVE WITH OFFSET may be used to shift a packed-decimal number an odd number of digit positions or to concatenate a sign to an unsigned packed-decimal number.

Assume that the three-byte unsigned packed-decimal number in storage locations 4500-4502 is to be moved to locations 5600-5603 and given the sign of the packed-decimal number ending at location 5603. Also assume:

- Register 12 contains 00 00 56 00.
- Register 15 contains 00 00 45 00.
- Storage locations 5600-5603 contain 77 88 99 0C.
- Storage locations 4500-4502 contain 12 34 56.

After the instruction:

```
Machine Format
Op Code L1 L2 B1 D1 B2 D2
F1 3 2 C 000 F 000

Assembler Format
Op Code D1(L1,B1),D2(L2,B2)
MVO 0(4,12),0(3,15)
```

is executed, the storage locations 5600-5603 contain 01 23 45 6C. Note that the second operand is extended on the left with one zero to fill out the first-operand field.
MOVE ZONES (MVZ)

The MOVE ZONES instruction can operate on overlapping or nonoverlapping fields, as can the instructions MOVE (MVC) and MOVE NUMERICS. When operating on nonoverlapping fields, MOVE ZONES works like the MOVE NUMERICS instruction (see its example), except that MOVE ZONES moves only the zone bits of each byte. To illustrate the use of MOVE ZONES with overlapping fields, assume that the following data field is in storage:

```
┌──┬──┬──┬──┬──┬──┐
│F1│C2│F3│C4│F5│C6│
└──┴──┴──┴──┴──┴──┘
```

Also assume that register 15 contains 00 00 08 00. The instruction:

```
MVZ 1(5,15),/zerodot(15)
```

propagates the zone bits from the byte at address 800 through the entire field, so that the field becomes:

```
800 805  
F1 C2 F3 C4 F5 C6
```

MULTIPLY (M, MR)

Assume that a number in register 5 is to be multiplied by the contents of a four-byte field at address 3750. Initially:

- The contents of register 4 are not significant.
- Register 5 contains 00 00 00 9A = 154₁₀ = the multiplicand.
- Register 11 contains 00 00 06 00.
- Register 12 contains 00 00 30 00.
- Storage locations 3750-3753 contain 00 00 00 83 = 131₁₀ = the multiplier.

The instruction required for performing the multiplication is:

```
M 4,X'15/zerodot'(11,12)
```

After the instruction is executed, the product is in the register pair 4 and 5:

- Register 4 contains 00 00 00 00.
- Register 5 contains 00 00 4E CE = 20,174₁₀.

Storage locations 3750-3753 are unchanged.

MULTIPLY HALFWORD (MH)

The MULTIPLY HALFWORD instruction is used to multiply the contents of a register by a two-byte field in storage. For example, assume that:

- Register 11 contains 00 00 00 15 =21₁₀ = the multiplicand.
- Register 14 contains 00 00 01 00.
- Register 15 contains 00 00 20 00.
Storage locations 2102-2103 contain FF D9 = -39₁₀ = the multiplier.

The instruction:

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R₁</th>
<th>X₂</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>4C</td>
<td>B</td>
<td>E</td>
<td>F</td>
<td>00</td>
</tr>
</tbody>
</table>

**Assembler Format**

Op Code  \( R₁, D₂(X₂, B₂) \)

\[ MH \quad 11,2(14,15) \]

multiplies the two numbers. The product, FF FF FC CD = -819₁₀, replaces the original contents of register 11.

Only the rightmost 32 bits of a product are stored in a register; any significant bits on the left are lost. No program interruption occurs on overflow.

**OR (O, OC, OI, OR)**

When the Boolean operator OR is applied to two bits, the result is one when either bit is one; otherwise, the result is zero. When two bytes are ORed, each pair of bits is handled separately; there is no connection from one bit position to another. The following is an example of ORing two bytes:

First-operand byte: 0011 010₁₀
Second-operand byte: 0101 110₀₀

Result byte: 0111 110₀₀

**OI Example**

A frequent use of the OR instruction is to set a particular bit to one. For example, assume that storage location 4891 contains 0100 001₀₀. To set the rightmost bit of this byte to one without affecting the other bits, the following instruction can be used (assume that register 8 contains 00 00 48 90):

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>I₂</th>
<th>B₁</th>
<th>D₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>01</td>
<td>8</td>
<td>00</td>
</tr>
</tbody>
</table>

**Assembler Format**

Op Code  \( D₁(B₁), I₂ \)

\[ OI \quad 1(8), X'01' \]

When this instruction is executed, the byte in storage is ORed with the immediate byte (the \( I₂ \) field of the instruction):

Location 4891: 0100 001₀₀
Immediate byte: 0000 001₀₀

Result: 0100 001₁₀

The resulting byte with bit 7 set to one is stored back in location 4891. Condition code 1 is set.

**PACK (PACK)**

Assume that storage locations 1000-1003 contain the following zoned-decimal number that is to be converted to a packed-decimal number and left in the same location:

\[ 1000 \quad 1003 \]

Also assume that register 12 contains 00 00 10 00. After the instruction:

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L₁</th>
<th>L₂</th>
<th>B₁</th>
<th>D₁</th>
<th>B₂</th>
<th>D₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>3</td>
<td>3</td>
<td>C</td>
<td>000</td>
<td>C</td>
<td>000</td>
</tr>
</tbody>
</table>

**Assembler Format**

Op Code  \( D₁(L₁,B₁), D₂(L₂,B₂) \)

\[ PACK \quad 0(4,12), 0(4,12) \]

is executed, the result in locations 1000-1003 is in the packed-decimal format:

\[ 1000 \quad 1003 \]

Packed number \[ 00 \quad 01 \quad 23 \quad 4C \]

**Notes:**

1. This example illustrates the operation of PACK when the first- and second-operand fields overlap completely.
2. During the operation, the second operand was extended on the left with zeros.
SEARCH STRING (SRST)

The SEARCH STRING instruction is used to search a second operand designated by general register R2 for a character specified in general register 0. The length of the second operand is known — the address of the first byte after the second operand is in general register R1.

When the specified character is found, condition code 1 is set, the address of the character is placed in general register R1, and the contents of general register R2 remain unchanged. When the address of the next second-operand byte to be examined equals the address in general register R1, condition code 2 is set, and the contents of general register R1 and R2 remain unchanged. When a CPU-determined number of second-operand bytes have been examined, condition code 3 is set, the address of the next byte to be processed in the second operand is placed in general register R2, and the contents of general register R1 remain unchanged.

SRST Example 1

Following is an example program that determines the end of string A, as indicated by an ending character equal to 00 hex (as in the C programming language), and then determines the address of the first character equal to C1 hex in the string. The program is based on the assumption that the second operand does not begin at location 0 or wrap around in storage, and, therefore, condition code 2 will not be set by the first SEARCH STRING instruction because of the address in general register 0. The program is not written for execution in the access-register mode.

SRST Example 2

Following is an example program that determines the address of the first character equal to C1 hex in the string A whose length is known. The program is not written for execution in the access-register mode.

SHIFT LEFT DOUBLE (SLDA)

The SHIFT LEFT DOUBLE instruction shifts the 63 numeric bits of an even-odd register pair to the left, leaving the sign bit unchanged. Thus, the instruction performs an algebraic left shift of a 64-bit signed binary integer.

For example, if the contents of registers 2 and 3 are:

```
00 7F 0A 72 FE DC BA 98 =
00000000 01111111 00001010 01110010
11111110 11011100 10111010 10011000
```

The instruction:

```
SLDA 2,31
```

results in registers 2 and 3 both being left-shifted 31 bit positions, so that their new contents are:

```
7F 6E 5D 4C 00 00 00 00 =
01111111 01101110 01011101 01001100
00000000 00000000 00000000 00000000
```
Because significant bits are shifted out of bit position 1 of register 2, overflow is indicated by setting condition code 3, and, if the fixed-point-overflow mask bit in the PSW is one, a fixed-point-overflow program interruption occurs.

SHIFT LEFT SINGLE (SLA)

The SHIFT LEFT SINGLE instruction is similar to SHIFT LEFT DOUBLE, except that it shifts only the 31 numeric bits of a single register. Therefore, this instruction performs an algebraic left shift of a 32-bit signed binary integer.

For example, if the contents of register 2 are:

\[
\text{zerodot} \ 7F \ \text{zerodot} \ A \ 72 = \text{zerodot} \ 7F \ \text{zerodot} \ A \ \text{zerodot} \ 72 = \text{zerodot} \ 1111111 \ \text{zerodot} \ 1111111 \ \text{zerodot} \ 1111111 \ \text{zerodot} \ 1111111
\]

The instruction:

Machine Format

\[
\begin{array}{cccc}
\text{Op Code} & R_1 & B_2 & D_2 \\
\hline
8B & 2 & /// & 0 & 008
\end{array}
\]

Assembler Format

\[
\text{Op Code} \ R_{basesuperone},D_{basesupertwo}(B_{basesupertwo})
\]

\[
\text{SLA} \ 2,8(0)
\]

results in register 2 being shifted left eight bit positions so that its new contents are:

\[
\text{zerodot} \ \text{zerodot} \ 7F \ \text{zerodot} \ A \ 72 = \text{zerodot} \ 1111111 \ \text{zerodot} \ 1111111 \ \text{zerodot} \ 1111111 \ \text{zerodot} \ 1111111
\]

Condition code 2 is set to indicate that the result is greater than zero.

If a left shift of nine places had been specified, a significant bit would have been shifted out of bit position 1. Condition code 3 would have been set to indicate this overflow and, if the fixed-point-overflow mask bit in the PSW were one, a fixed-point overflow interruption would have occurred.

STORE CHARACTERS UNDER MASK (STCM)

STORE CHARACTERS UNDER MASK (STCM) may be used to place selected bytes from a register into storage. For example, if it is desired to store a three-byte address from general register 8 into location FIELD3, assume:

Machine Format

\[
\begin{array}{cccc}
\text{Op Code} & R_1 & M_3 & S_2 \\
\hline
\text{BE} & 8 & 7 & \text{* * * *}
\end{array}
\]

Register Format

\[
\begin{array}{cccc}
\text{Op Code} & R_1,M_3,S_2 \\
\hline
\text{STCM} & 8,B\ '0111\ ',\text{FIELD3}
\end{array}
\]

Register 8: 12 34 56 78
FIELD3 (before): not significant
FIELD3 (after): 34 56 78

As another example:

Machine Format

\[
\begin{array}{cccc}
\text{Op Code} & R_1 & M_3 & S_2 \\
\hline
\text{BE} & 9 & 5 & \text{* * * *}
\end{array}
\]

Register Format

\[
\begin{array}{cccc}
\text{Op Code} & R_1,M_3,S_2 \\
\hline
\text{STCM} & 9,B\ '0101\ ',\text{FIELD2}
\end{array}
\]

Register 9: 01 23 45 67
FIELD2 (before): not significant
FIELD2 (after): 23 67

STORE MULTIPLE (STM)

Assume that the contents of general registers 14, 15, 0, and 1 are to be stored in consecutive four-byte fields starting with location 4050 and that:

- Register 14 contains 00 00 25 63.
- Register 15 contains 00 01 27 36.
- Register 0 contains 12 43 00 62.
- Register 1 contains 73 26 12 57.
- Register 6 contains 00 00 40 00.

The initial contents of locations 4050-405F are not significant.

The STORE MULTIPLE instruction allows the use of just one instruction to store the contents of the four registers:

Machine Format

\[
\begin{array}{cccc}
\text{Op Code} & R_1 & R_3 & B_2 & D_2 \\
\hline
90 & E & 1 & 6 & 050
\end{array}
\]
Assembler Format
Op Code   R₁,R₃,Dₓ(B₂)
STM   14,1,X'50'(6)

After the instruction is executed:
Locations 4050-4053 contain 00 00 25 63.
Locations 4054-4057 contain 00 01 27 36.
Locations 4058-405B contain 12 43 00 62.
Locations 405C-405F contain 73 26 12 57.

TEST UNDER MASK (TM)

The TEST UNDER MASK instruction examines selected bits of a byte and sets the condition code accordingly. For example, assume that:

Storage location 9999 contains FB.
Register 7 contains 00 00 99 90.

Assume the instruction to be:

Machine Format
Op Code   I₂ B₁ D₁

<table>
<thead>
<tr>
<th>91</th>
<th>C3</th>
<th>7</th>
<th>009</th>
</tr>
</thead>
</table>

Assembler Format
Op Code   D₁(B₁),I₂

TM   9(7),B'11000011'

The instruction tests only those bits of the byte in storage for which the mask bits are ones:

FB   = 1111 1011₂
Mask   = 1100 0011₂

Test   = 11xx xx11₂
Condition code 3 is set: all selected bits in the test result are ones. (The bits marked “x” are ignored.)

If location 9999 had contained B9, the test would have been:

B9   = 1011 1001₂
Mask   = 1100 0011₂

Test   = 10xx xx01₂
Condition code 1 is set: the selected bits are both zeros and ones.

If location 9999 had contained 3C, the test would have been:

3C   = 0011 1100₂
Mask   = 1100 0011₂

Test   = 00xx xx00₂
Condition code 0 is set: all selected bits are zeros.

Note: Storage location 9999 remains unchanged.

TRANSLATE (TR)

The TRANSLATE instruction can be used to translate data from any character code to any other desired code, provided that each character code consists of eight bits or fewer. An appropriate translation table is required in storage.

In the following example, EBCDIC code is translated to ASCII code. The first step is to create a 256-byte table in storage locations 1000-10FF. This table contains the characters of the ASCII code in the sequence of the binary representation of the EBCDIC code; that is, the ASCII representation of a character is placed in storage at the starting address of the table plus the binary value of the EBCDIC representation of the same character.

For simplicity, the example shows only the part of the table containing the decimal digits:

<table>
<thead>
<tr>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
</table>

Assume that the four-byte field at storage location 2100 contains the EBCDIC code for the digits 1984:

Locations 2100-2103 contain F1 F9 F8 F4.
Register 12 contains 00 00 21 00.
Register 15 contains 00 00 10 00.

As the instruction:

Machine Format
Op Code   L B₁ D₁ B₂ D₂

<table>
<thead>
<tr>
<th>DC</th>
<th>03</th>
<th>C</th>
<th>000</th>
<th>F</th>
<th>000</th>
</tr>
</thead>
</table>

Appendix A. Number Representation and Instruction-Use Examples A-31
is executed, the binary value of each EBCDIC byte is added to the starting address of the table, and the resulting address is used to fetch an ASCII byte:

Table starting address: 1000
First EBCDIC byte: F1
Address of ASCII byte: 10F1

After execution of the instruction:

Locations 2100-2103 contain 31 39 38 34.

Thus, the ASCII code for the digits 1984 has replaced the EBCDIC code in the four-byte field at storage location 2100.

**TRANSLATE AND TEST (TRT)**

The TRANSLATE AND TEST instruction can be used to scan a data field for characters with a special meaning. To indicate which characters have a special meaning, a table similar to the one used for the TRANSLATE instruction is set up, except that zeros in the table indicate characters without any special meaning and nonzero values indicate characters with a special meaning.

Figure A-4 has been set up to distinguish alphanumeric characters (A to Z and 0 to 9) from blanks, certain special symbols, and all other characters which are considered invalid. EBCDIC coding is assumed. The 256-byte table is assumed stored at locations 2000-20FF.

**Assembler Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>D₁(L₁,B₁),D₂(B₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR</td>
<td>0(4,12),0(15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address of ASCII byte: 10F1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Table starting address: 1000</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>First EBCDIC byte: F1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>After execution of the instruction:</th>
</tr>
</thead>
</table>

Locations 2100-2103 contain 31 39 38 34.

Thus, the ASCII code for the digits 1984 has replaced the EBCDIC code in the four-byte field at storage location 2100.

**Note:** If the character codes in the statement being translated occupy a range smaller than 00 through FF₁₆, a table of fewer than 256 bytes can be used.

**Figure A-4. Translate and Test Table**

The table entries for the alphameric characters in EBCDIC are 00; thus, the letter A (code C1) corresponds to byte location 20C1, which contains 00.

The 15 special symbols have nonzero entries from 04₁₆ to 3C₁₆ in increments of 4. Thus, the blank (code 40) has the entry 04₁₆, the period (code 4B) has the entry 08₁₆, and so on.

All other table positions have the entry 40₁₆ to indicate an invalid character.

The table entries are chosen so that they may be used to select one of a list of 16 words containing addresses of different routines to be entered for each special symbol or invalid character encountered during the scan.

Assume that this list of 16 branch addresses is stored at locations 3004-3043.

Starting at storage location CA80, there is the following sequence of 21₁₆ EBCDIC characters, where “b” stands for a blank.

---

0 1 2 3 4 5 6 7 8 9 A B C D E F
200 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
201 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
202 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
203 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
204 04 40 40 40 40 40 40 40 40 40 40 40 40 40 40
205 14 40 40 40 40 40 40 40 40 40 40 40 40 40 40
206 24 28 40 40 40 40 40 40 40 40 40 40 40 40 40
207 40 40 40 40 40 40 40 40 40 40 30 34 38 3C 40
208 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
209 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
20A 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
20B 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
20C 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
20D 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
20E 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
20F 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
Locations CA80-CA94:
UNPKbPROUT(9),WORD(5)

Also assume:
Register 1 contains 00 00 CA 7F.
Register 2 contains 00 00 30 00.
Register 15 contains 00 00 20 00.

As the instruction:

Machine Format
Op Code L B/basesuperone D/basesuperone B/basesupertwo D/basesupertwo

Asm. Format
Op Code D/basesuperone(L,basesuperone),D/basesupertwo(B/basesupertwo)

is executed, the value of the first source byte, the EBCDIC code for the letter U, is added to the starting address of the table to produce the address of the table entry to be examined:

Table starting address 2000
First source byte (U) E4

Address of table entry 20E4

Because zeros were placed in storage location 20E4, no special action occurs. The operation continues with the second and subsequent source bytes until it reaches the blank in location CA84. When this symbol is reached, its value is added to the starting address of the table, as usual:

Table starting address 2000
Source byte (blank) 40

Address of table entry 2040

Because location 2040 contains a nonzero value, the following actions occur:

- The address of the source byte, 00CA84, is placed in the rightmost 24 bits of register 1.
- The table entry, 04, is placed in the rightmost eight bits of register 2, which now contains 00 00 30 04.
- Condition code 1 is set (scan not completed).

The TRANSLATE AND TEST instruction may be followed by instructions to branch to the routine at the address found at location 3004, which corresponds to the blank character encountered in the scan. When this routine is completed, program control may return to the TRANSLATE AND TEST instruction to continue the scan, except that the length must first be adjusted for the characters already scanned.

For this purpose, the TRANSLATE AND TEST may be executed by the use of an EXECUTE instruction, which supplies the length specification from a general register. In this way, a complete statement scan can be performed with a single TRANSLATE AND TEST instruction used repeatedly by means of EXECUTE, and without modifying any instructions in storage. In the example, after the first execution of TRANSLATE AND TEST, register 1 contains the address of the last source byte translated. It is then a simple matter to subtract this address from the address of the last source byte (CA94) to produce a length specification. This length minus one is placed in the register that is referenced as the R1 field of the EXECUTE instruction. (Note that the length code in the machine format is one less than the total number of bytes in the field.) The second-operand address of the EXECUTE instruction points to the TRANSLATE AND TEST instruction, which is the same as illustrated above, except for the length (L) which is set to zero.

UNPACK (UNPK)

Assume that storage locations 2501-2502 contain a signed, packed-decimal number that is to be unpacked and placed in storage locations 1000-1004. Also assume:

Register 12 contains 00 00 10 00.
Register 13 contains 00 00 25 00.
Storage locations 2501-2502 contain 12 3D.
The initial contents of storage locations 1000-1004 are not significant.

After the instruction:

Machine Format
Op Code L L2 B1 D1 B2 D2

Asm. Format
Op Code D1(L,B1),D2(L2,B2)

UNPK 0(5,12),1(2,13)
is executed, the storage locations 1000-1004 contain F0 F0 F1 F2 D3.

**UPDATE TREE (UPT)**

See "Sorting Instructions" on page A-51.

---

**Decimal Instructions**

(See Chapter 8, "Decimal Instructions" for a complete description of the decimal instructions.)

---

**ADD DECIMAL (AP)**

Assume that the signed, packed-decimal number at storage locations 500-503 is to be added to the signed, packed-decimal number at locations 2000-2002. Also assume:

- Register 12 contains 00 00 20 00.
- Register 13 contains 00 00 05 00.
- Storage locations 2000-2002 contain 38 46 0D (a negative number).
- Storage locations 500-503 contain 01 12 34 5C (a positive number).

After the instruction:

- Machine Format
  ```plaintext
  ┌────────┬────┬────┬────┬────┬────┐
  │ FA │ 2 │ 3 │ C │ /zerodot/zerodot/zerodot│
  └────────┴────┴────┴────┴────┘
  ┌────────┬────┬────┬────┬────┬────┐
  │ D │ /zerodot/zerodot/zerodot│ D │ /zerodot/zerodot/zerodot│
  └────────┴────┴────┴────┴────┘
  ``
- Assembler Format
  ```plaintext
  AP /zerodot(3,12),/zerodot(4,13)
  ```

---

**COMPARE DECIMAL (CP)**

Assume that the signed, packed-decimal contents of storage locations 700-703 are to be algebraically compared with the signed, packed-decimal contents of locations 500-502. Also assume:

- Register 12 contains 00 00 06 00.
- Register 13 contains 00 00 03 00.
- Storage locations 700-703 contain 17 25 35 6D.
- Storage locations 500-502 contain 72 14 2D.

After the instruction:

- Machine Format
  ```plaintext
  ┌────────┬────┬────┬────┬────┬────┐
  │ F9 │ 3 │ 2 │ C │ 1/zerodot/zerodot│ D │ 2/zerodot/zerodot│
  └────────┴────┴────┴────┴────┴────┘
  ``
- Assembler Format
  ```plaintext
  CP X'1/zerodot/zerodot'(4,12),X'2/zerodot/zerodot'(3,13)
  ```

---

**DIVIDE DECIMAL (DP)**

Assume that the signed, packed-decimal number at storage locations 2000-2004 (the dividend) is to be divided by the signed, packed-decimal number at locations 3000-3001 (the divisor). Also assume:

- Register 12 contains 00 00 20 00.
- Register 13 contains 00 00 30 00.
- Storage locations 2000-2004 contain 01 23 45 67 8C.
- Storage locations 3000-3001 contain 32 1D.

After the instruction:

- Machine Format
  ```plaintext
  ┌────────┬────┬────┬────┬────┬────┐
  │ FD │ 4 │ 1 │ C │ /zerodot/zerodot/zerodot│
  └────────┴────┴────┴────┴────┘
  ``
- Assembler Format
  ```plaintext
  DP (L1,B1),D2(L2,B2)
  ```

is executed, condition code 1 is set, indicating that the first operand (the contents of locations 700-703) is less than the second.
Assembler Format
Op Code  D₁(L₁,B₁),D₂(L₂,B₂)

DP  0(5,12),0(2,13)

is executed, the dividend is entirely replaced by the signed quotient and remainder, as follows:

Locations 2000-2004

| 38 | 46 | 0D | 01 | 8C |

is executed, the dividend is entirely replaced by the signed quotient and remainder, as follows:

Notes:
1. Because the dividend and divisor have different signs, the quotient receives a negative sign.
2. The remainder receives the sign of the dividend and the length of the divisor.
3. If an attempt were made to divide the dividend by the one-byte field at location 3001, the quotient would be too long to fit within the four bytes allotted to it. A decimal-divide exception would exist, causing a program interruption.

EDIT (ED)

Before decimal data in the packed format can be used in a printed report, digits and signs must be converted to printable characters. Moreover, punctuation marks, such as commas and decimal points, may have to be inserted in appropriate places. The highly flexible EDIT instruction performs these functions in a single instruction execution.

This example shows step-by-step one way that the EDIT instruction can be used. The field to be edited (the source) is four bytes long; it is edited against a pattern 13 bytes long. The following symbols are used:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>(Hexadecimal 40) Blank character</td>
</tr>
<tr>
<td>(</td>
<td>(Hexadecimal 21) Significance starter</td>
</tr>
<tr>
<td>d</td>
<td>(Hexadecimal 20) Digit selector</td>
</tr>
</tbody>
</table>

Assume that register 12 contains:

| 00 | 00 | 10 | 00 |

and that the source and pattern fields are:

Source

| 1200 | 1203 |
| 02 | 57 | 42 | 6C |

Pattern

| 1000 | 100C |
| 40 | 20 | 20 | 6B | 20 | 21 | 20 | 4B | 20 | 20 | 40 | C3 | D9 |

Execution of the instruction:

Machine Format
Op Code  L B₁ D₁ B₂ D₂

| DE | 0C | C | 000 | C | 200 |

Assembler Format
Op Code  D₁(L₁,B₁),D₂(B₂)

ED  0(13,12),X’200’(12)

alters the pattern field as follows:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Digit</th>
<th>Significance Indicator (Before/After)</th>
<th>Rule</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0</td>
<td>off/off</td>
<td>fill</td>
<td>bdd,d(d.ddbCR</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>off/on(2)</td>
<td>digit</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>,</td>
<td>5</td>
<td>on/on</td>
<td>leave</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>(</td>
<td>7</td>
<td>on/on</td>
<td>digit</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>on/on</td>
<td>digit</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>,</td>
<td>2</td>
<td>on/on</td>
<td>leave</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>d</td>
<td>6+</td>
<td>on/off(3)</td>
<td>digit</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>off/off</td>
<td>fill</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>off/off</td>
<td>fill</td>
<td>bbd,d(d.ddbCR</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td>off/off</td>
<td>fill</td>
<td>bbd,d(d.ddbCR</td>
</tr>
</tbody>
</table>

Notes:
1. This character is the fill byte.
2. First nonzero decimal source digit turns on significance indicator.
3. Plus sign in the four rightmost bits of the byte turns off significance indicator.
Thus, after the instruction is executed, the pattern field contains the result as follows:

```
Pattern
1000     100C
  40 40 F2 6B F5 F7 4B F2 F6 40 40 40
  b b 2 , 5 7 4 . 2 6 b b b
```

This pattern field prints as:

```
2,574.26
```

The source field remains unchanged. Condition code 2 is set because the number was greater than zero.

If the number in the source field is changed to the negative number 00 00 02 6D and the original pattern is used, the edited result this time is:

```
Pattern
1000     100C
  40 40 40 40 40 40 F0 4B F2 F6 40 C3 D9
  b b b b b b $ /zerodot . 2 6 b C R
```

This pattern field prints as:

```
$/zerodot.26 CR
```

Condition code 1 is set because the number is less than zero.

**EDIT AND MARK (EDMK)**

The EDIT AND MARK instruction may be used, in addition to the functions of EDIT, to insert a currency symbol, such as a dollar sign, at the appropriate position in the edited result. Assume the same source in storage locations 1200-1203, the same pattern in locations 1000-100C, and the same contents of general register 12 as for the EDIT instruction above. The previous contents of general register 1 (GR1) are not significant; a LOAD ADDRESS instruction is used to set up the first digit position that is forced to print if no significant digits occur to the left.

The instructions:

- `LA 1,6(0,12)` Load address of forced significant digit into GR1
- `EDMK 0(13,12),X'200'(12)` Leave address of first significant digit in GR1
- `BCTR 1,0` Subtract 1 from address in GR1
- `MVI 0(1),C'$` Store dollar sign at address in GR1

produce the following results for the two examples under EDIT:

```
Pattern
1000     100C
  40 5B F2 6B F5 F7 4B F2 F6 40 40 40
  b $ 2 , 5 7 4 . 2 6 b b b
```

This pattern field prints as:

```
$2,574.26
```

Condition code 2 is set to indicate that the number edited was greater than zero.

```
Pattern
1000     100C
  40 40 40 40 40 40 5B F0 4B F2 F6 40 C3 D9
  b b b b b b $ 0 . 2 6 b C R
```

This pattern field prints as:

```
$0.26 CR
```

Condition code 1 is set because the number is less than zero.

**MULTIPLY DECIMAL (MP)**

Assume that the signed, packed-decimal number in storage locations 1202-1204 (the multiplicand) is to be multiplied by the signed, packed-decimal number in locations 500-501 (the multiplier).

```
1202 1204
Multiplicand 38 46 6D
```
The multiplicand must first be extended to have at least two bytes of leftmost zeros, corresponding to the multiplier length, so as to avoid a data exception during the multiplication. ZERO AND ADD can be used to move the multiplicand into a longer field. Assume:

- Register 4 contains 00 00 12 00.
- Register 6 contains 00 00 05 00.

Then execution of the instruction:

```
ZAP X'1/zerodot/zerodot'(5,4),2(3,4)
```

sets up a new multiplicand in storage locations 1300-1304:

```
13/zerodot/zerodot 13/zerodot4
```

Now, after the instruction:

```
MP X'1/zerodot/zerodot'(5,4),/zerodot(2,6)
```

is executed, storage locations 1300-1304 contain the product: 01 23 45 66 0C.

### SHIFT AND ROUND DECIMAL (SRP)

The SHIFT AND ROUND DECIMAL (SRP) instruction can be used for shifting decimal numbers in storage to the left or right. When a number is shifted right, rounding can also be done.

#### Decimal Left Shift

In this example, the contents of storage location FIELD1 are shifted three places to the left, effectively multiplying the contents of FIELD1 by 1000. FIELD1 is six bytes long. The following instruction performs the operation:

```
SRP FIELD1(6),3,0
```

FIELD1 (before): 00 01 23 45 67 8C
FIELD1 (after): 12 34 56 78 00 0C

The second-operand address in this instruction specifies the shift amount (three places). The rounding digit, I, is not used in a left shift, but it must be a valid decimal digit. After execution, condition code 2 is set to show that the result is greater than zero.

#### Decimal Right Shift

In this example, the contents of storage location FIELD2 are shifted one place to the right, effectively dividing the contents of FIELD2 by 10 and discarding the remainder. FIELD2 is five bytes in length. The following instruction performs this operation:

```
SRP FIELD2(5),64-1,0
```

FIELD 2 (before): 01 23 45 67 8C
FIELD 2 (after): 00 12 34 56 7C

The 6-bit two's complement for -1 is 011111.
In the SRP instruction, shifts to the right are specified in the second-operand address by negative shift values, which are represented as a six-bit value in two's complement form.

The six-bit two's complement of a number, \( n \), can be specified as \( 64 - n \). In this example, a right shift of one is represented as \( 64 - 1 \).

Condition code 2 is set.

**Decimal Right Shift and Round**

In this example, the contents of storage location FIELD3 are shifted three places to the right and rounded, in effect dividing by 1000 and rounding up. FIELD3 is four bytes in length.

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L1</th>
<th>I3</th>
<th>S1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>3</td>
<td>5</td>
<td>****</td>
<td>0</td>
<td>03D</td>
</tr>
</tbody>
</table>

The shift amount (three places) is specified in the \( D_2 \) field. The \( I_3 \) field specifies a rounding digit of 5. The rounding digit is added to the last digit shifted out (which is a 6), and the carry is propagated to the left. The sign is ignored during the addition.

Condition code 1 is set because the result is less than zero.

**Multiplying by a Variable Power of 10**

Since the shift value specified by the SRP instruction specifies both the direction and amount of the shift, the operation is equivalent to multiplying the decimal first operand by 10 raised to the power specified by the shift value.

If the shift value is to be variable, it may be specified by the \( B_2 \) field instead of the displacement \( D_2 \) of the SRP instruction. The general register designated by \( B_2 \) should contain the shift value (power of 10) as a signed binary integer.

A fixed scale factor modifying the variable power of 10 may be specified by using both the \( B_2 \) field (variable part in a general register) and the \( D_2 \) field (fixed part in the displacement).

The SRP instruction uses only the rightmost six bits of the effective address \( D_2(B_2) \) and interprets them as a six-bit signed binary integer to control the left or right shift as in the preceding shift examples.

**ZERO AND ADD (ZAP)**

Assume that the signed, packed-decimal number at storage locations 4500-4502 is to be moved to locations 4000-4004 with four leading zeros in the result field. Also assume:

- Register 9 contains 00 00 40 00.
- Storage locations 4000-4004 contain 12 34 56 78 90.
- Storage locations 4500-4502 contain 38 46 0D.

After the instruction:

**Machine Format**

<table>
<thead>
<tr>
<th>Op Code</th>
<th>L1</th>
<th>L2</th>
<th>B1</th>
<th>D1</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>000</td>
<td>9</td>
<td>500</td>
</tr>
</tbody>
</table>

The ZAP instruction is executed, the storage locations 4000-4004 contain 00 00 38 46 0D; condition code 1 is set to indicate a negative result without overflow.
Note that, because the first operand is not checked for valid sign and digit codes, it may contain any combination of hexadecimal digits before the operation.

Hexadecimal-Floating-Point Instructions

(See Chapter 9, “Floating-Point Overview and Support Instructions” for a complete description of the hexadecimal-floating-point instructions.)

In this section, the abbreviations FPR0, FPR2, FPR4, and FPR6 stand for floating-point registers 0, 2, 4, and 6 respectively.

**ADD NORMALIZED (AD, ADR, AE, AER, AXR)**

The ADD NORMALIZED instruction performs the addition of two HFP numbers and places the normalized result in a floating-point register. Neither of the two numbers to be added must necessarily be in normalized form before addition occurs. For example, assume that:

- FPR6 contains the unnormalized number C3 08 21 00 00 00 00 00 = -82.16 = -130.06\(10\) approximately.
- Storage locations 2000-2007 contain the normalized number 41 12 34 56 00 00 00 00 = +1.23456\(10\) = +1.14\(2\) approximately.
- Register 13 contains 00 00 20 00.

The instruction:

Machine Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7A</td>
<td>6</td>
<td>0</td>
<td>D</td>
<td>000</td>
</tr>
</tbody>
</table>

Assembler Format

Op Code R1, D2(X2, B2)

AE 6,0(0,13)

performs the short-precision addition of the two operands, as follows.

The characteristics of the two numbers (43 and 41) are compared. Since the number in storage has a characteristic that is smaller by 2, it is right-shifted two hexadecimal digit positions. One guard digit is retained on the right. The fractions of the two numbers are then added algebraically:

<table>
<thead>
<tr>
<th>Fraction GD(1)</th>
<th>FPR6</th>
<th>Shifted number from storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-43 08 21 00</td>
<td>+43 00 12 34 5</td>
</tr>
</tbody>
</table>

Intermediate sum -43 08 OΕ CB B

Left-shifted sum -42 8/érieur EB

Guard digit

Because the intermediate sum is unnormalized, it is left-shifted to form the normalized HFP number -80.ECB\(16\) = -128.92\(10\) approximately. Combining the sign with the characteristic, the result is C2 80 EC BB, which replaces the left half of FPR6. The right half of FPR6 and the contents of storage locations 2000-2007 are unchanged. Condition code 1 is set to indicate a result less than zero.

If the long-precision instruction AD were used, the result in FPR6 would be C2 80 EC BA A0 00 00 00. Note that use of the long-precision instruction would avoid a loss of precision in this example.

**ADD UNNORMALIZED (AU, AUR, AW, AWR)**

The ADD UNNORMALIZED instruction operates the same as the ADD NORMALIZED instruction, except that the final result is not normalized. For example, using the the same operands as in the example for ADD NORMALIZED, when the short-precision instruction:

Machine Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>X2</th>
<th>B2</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7E</td>
<td>6</td>
<td>0</td>
<td>D</td>
<td>000</td>
</tr>
</tbody>
</table>

Assembler Format

Op Code R1, D2(X2, B2)

AU 6,0(0,13)

is executed, the two numbers are added as follows:
The guard digit participates in the addition but is discarded. The unnormalized sum replaces the left half of FPR6. Condition code 1 is set because the result is less than zero.

The truncated result in FPR6 (C3 08 0E CB 00 00 00 00) shows a loss of a significant digit when compared to the result of short-precision normalized addition.

**COMPARE (CD, CDR, CE, CER)**

Assume that FPR4 contains 43 00 00 00 00 00 00 00 (zero), and FPR6 contains 35 12 34 56 78 9A BC DE (a positive number). The contents of the two registers are to be compared using a long-precision COMPARE instruction.

Machine Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Assembler Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1, R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDR</td>
<td>4,6</td>
</tr>
</tbody>
</table>

The number with the smaller characteristic, which is in register FPR6, is right-shifted 43 - 35 hex (67 - 53 decimal) or 14 digit positions, so that the two characteristics agree. The shifted number is 43 00 00 00 00 00 00 00, with a guard digit of one. Therefore, when the two numbers are compared, condition code 1 is set, indicating that operand 1 in FPR4 is less than operand 2 in FPR6.

If the example is changed to a second operand with a characteristic of 34 instead of 35, so that FPR6 contains 34 12 34 56 78 9A BC DE, the operand is right-shifted 15 positions, leaving all fraction digits and the guard digit as zeros. Condition code 0 is set, indicating equality. This example shows that two HFP numbers with different characteristics or fractions may compare equal if the numbers are unnormalized or zero.

As another example of comparing unnormalized HFP numbers, 41 00 12 34 56 78 9A BC compares equal to all numbers of the form 3F 12 34 56 78 9A BC 0X (X represents any hexadecimal digit). When the COMPARE instruction is executed, the two rightmost digits are shifted right two places, the 0 becomes the guard digit, and the X does not participate in the comparison.

However, when two normalized HFP numbers are compared, the relationship between numbers that compare equal is unique: each digit in one number must be the same as the corresponding digit in the other number.

**DIVIDE (DD, DDR, DE, DER)**

Assume that the first operand (the dividend) is in FPR2 and the second operand (the divisor) in FPR0. If the operands are in the short-precision format, the resulting quotient is returned to FPR2 by the instruction:

Machine Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Assembler Format

<table>
<thead>
<tr>
<th>Op Code</th>
<th>R1, R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DER</td>
<td>2,0</td>
</tr>
</tbody>
</table>

Several examples of short-precision HFP division, with the dividend in FPR2 and the divisor in FPR0, are shown below. For case A, the result, which replaces the dividend, is obtained in the following steps.
MULTIPLY (MD, MDR, MDE, MDER, MXD, MXDR, MXR)

For this example, the following long-precision operands are in FPR0 and FPR2:

FPR0: -33 606060 60606060
FPR2: -5A 200000 20000020

A long-precision product is generated by the instruction:

Machine Format
Op Code R₁ R₂
2C 0 2

Assembler Format
Op Code R₁,R₂
MDR 0,2

If the operands were not already normalized, the instruction would first normalize them. It then generates an intermediate result consisting of the full 28-digit hexadecimal product fraction obtained by multiplying the 14-digit hexadecimal operand fractions, together with the appropriate sign and a characteristic that is the sum of the operand characteristics less 64 (40 hex):

The fraction multiplication is performed as follows:

\[
\times .2/zerodot/zerodot/zerodot/zerodot/zerodot2/zerodot/zerodot/zerodot/zerodot/zerodot2/zerodot
\]

Attaching the sign and characteristic to the fraction gives:

+4D C0C0C0C0C0C0C0

Because this intermediate product has a leading zero, it is then normalized. The truncated final result placed in FPR0 is:

+4C C0C0C1 81818241

HALVE (HDR, HER)

HALVE produces the same result as HFP DIVIDE with a divisor of 2.0. Assume FPR2 contains the long-precision number +48 30 00 00 00 00 00 0F. The following HALVE instruction produces the result +48 18 00 00 00 00 00 07 in FPR2:

Machine Format
Op Code R₁ R₂
24 2 2

Assembler Format
Op Code R₁,R₂
HDR 2,2

Case C shows a number being divided by 4.0. Case D divides the same number by 2.0, and case E divides the result of case D again by 2.0. The results of cases C and E differ in the rightmost hexadecimal digit position, which illustrates an effect of result truncation.
**Hexadecimal-Floating-Point-Number Conversion**

The following examples illustrate one method of converting between binary fixed-point numbers (32-bit signed binary integers) and normalized HFP numbers. Conversion must provide for the different representations used with negative numbers: the two's-complement form for signed binary integers, and the signed-absolute-value form for the fractions of HFP numbers.

### Fixed Point to Hexadecimal Floating Point

The method used here inverts the leftmost bit of the 32-bit signed binary integer, which is equivalent to adding $2^{31}$ to the number and considering the result to be positive. This changes the number from a signed integer in the range $2^{31} - 1$ through $-2^{31} - 1$ to an unsigned integer in the range $2^{32} - 1$ through 0. After conversion to the long HFP format, the value $2^{31}$ is subtracted again.

Assume that general register 9 (GR9) contains the integer -59 in two's-complement form:

```
GR9: FF FF FF C5
```

Further, assume two eight-byte fields in storage: TEMP, for use as temporary storage, and TWO31, which contains the floating-point constant $2^{31}$ in the following format:

```
TWO31: 4E 00 00 00 00 00 00 00
```

This is an unnormalized long HFP number with the characteristic 4E, which corresponds to a radix point (hexadecimal point) to the right of the number.

The following instruction sequence performs the conversion:

```
X 9,TWO31+4
ST 9,TEMP+4
MVC TEMP(4),TWO31
LD 2,TEMP
SD 2,TWO31
```

### Hexadecimal Floating Point to Fixed Point

The procedure described here consists basically in reversing the steps of the previous procedure. Two additional considerations must be taken into account. First: the HFP number may not be an exact integer. Truncating the excess hexadecimal digits on the right requires shifting the number one digit position farther to the right than desired for the final result, so that the units digit occupies the position of the guard digit. Second: the HFP number may have to be tested as to whether it is outside the range of numbers representable as a 32-bit signed binary integer.

Assume that floating-point register 6 contains the number 59.25_16 = 3B.416 in normalized form:

```
FPR6: 42 3B 40 00 00 00 00 00
```

Further, assume three eight-byte fields in storage: TEMP, for use as temporary storage, and the constants $2^{32}$ (TWO32) and $2^{31}$ (TWO31R) in the following formats:

```
TWO32: 4E 00 00 01 00 00 00 00
TWO31R: 4F 00 00 08 00 00 00 00
```

The constant TWO31R is shifted right one more position than the constant TWO31 of the previous example, so as to force the units digit into the guard-digit position.

The following instruction sequence performs the integer truncation, range tests, and conversion to a signed binary integer in general register 8 (GR8):
The **SUBTRACT NORMALIZED** (SD) instruction shifts the fraction of the number to the right until it lines up with **TWO31R**, which causes the fraction digit 4 to fall to the right of the guard digit and be lost; the result of subtracting $2^{31}$ from the remaining digits is renormalized. The result should be less than zero; if not, the original number was too large in the positive direction. The first **BRANCH ON CONDITION** (BC) performs this test.

The **ADD UNNORMALIZED** (AW) instruction adds $2^{32} - 2^{31}$ to correct for the previous subtraction and another $2^{31}$ to change to an all-positive range. The second BC tests for a result less than zero, showing that the original number was too large in the negative direction. The unnormalized result is placed in temporary storage by the **STORE** (STD) instruction. There the leftmost bit of the binary integer is inverted by the **EXCLUSIVE OR** (XI) instruction to subtract $2^{31}$ and thus convert the unsigned number to the signed format. The final result is loaded into GR8.

#### Multiprogramming and Multiprocessing Examples

When two or more programs sharing common storage locations are being executed concurrently in a multiprogramming or multiprocessing environment, one program may, for example, set a flag bit in the common-storage area for testing by another program. It should be noted that the instructions **AND** (NI or NC), **EXCLUSIVE OR** (XI or XC), and **OR** (OI or OC) could be used to set flag bits in a multiprogramming environment; but the same instructions may cause program logic errors in a multiprocessing configuration where two or more CPUs can fetch, modify, and store data in the same storage locations simultaneously.

### Example of a Program Failure Using OR Immediate

Assume that two independent programs try to set different bits to one in a common byte in storage. The following example shows how the use of the instruction **OR immediate** (OI) can fail to accomplish this, if the programs are executed simultaneously on two different CPUs. One of the possible error situations is depicted.

<table>
<thead>
<tr>
<th>Execution of instruction</th>
<th>FLAGS</th>
<th>Execution of instruction</th>
<th>FLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>OI FLAGS, X'01' on CPU A</td>
<td></td>
<td>Fetch FLAGS X'00' on CPU B</td>
<td></td>
</tr>
<tr>
<td>Fetch FLAGS X'00'</td>
<td>X'00'</td>
<td>OR X'80' into X'00'</td>
<td></td>
</tr>
<tr>
<td>OR X'01' into X'00'</td>
<td>X'00'</td>
<td>Store X'80' into FLAGS</td>
<td></td>
</tr>
<tr>
<td>Store X'01' into FLAGS</td>
<td>X'01'</td>
<td>FLAGS should have value of X'81' following both updates.</td>
<td></td>
</tr>
</tbody>
</table>

The problem shown here is that the value stored by the OI instruction executed on CPU A overlays the value that was stored by CPU B. The X'80' flag bit was erroneously turned off, and the data is now invalid.

The **COMPARE AND SWAP** instruction has been provided to overcome this and similar problems.

<table>
<thead>
<tr>
<th>SD 6,TWO31R</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR6: C87F FFFF C500 0000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BC 11,OVERFLOW</th>
<th>Branch to overflow routine if result is greater than or equal to zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR6: 4E00 0000 8000 003B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AW 6,TWO32</th>
<th>Branch to overflow routine if result is less than zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPR6: 4E00 0000 8000 003B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BC 4,OVERFLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch to overflow routine if result is less than zero</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STD 6,TEMP</th>
<th>TEMP: 4E00 0000 8000 003B</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>XI TEMP+4,X'80'</th>
<th>TEMP: 4E00 0000 0000 003B</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>L 8,TEMP+4</th>
<th>GR8: 0000 003B</th>
</tr>
</thead>
</table>

Appendix A. Number Representation and Instruction-Use Examples
Conditional Swapping Instructions (CS, CDS)

The COMPARE AND SWAP (CS) and COMPARE DOUBLE AND SWAP (CDS) instructions can be used in multiprogramming or multiprocessing environments to serialize access to counters, flags, control words, and other common storage areas.

The following examples of the use of the COMPARE AND SWAP and COMPARE DOUBLE AND SWAP instructions illustrate the applications for which the instructions are intended. It is important to note that these are examples of functions that can be performed by programs while the CPU is enabled for interruption (multiprogramming) or by programs that are being executed in a multiprocessing configuration. That is, the routine allows a program to modify the contents of a storage location while the CPU is enabled, even though the routine may be interrupted by another program on the same CPU that will update the location, and even though the possibility exists that another CPU may simultaneously update the same location.

The COMPARE AND SWAP instruction first checks the value of a storage location and then modifies it only if the value is what the program expects; normally this would be a previously fetched value. If the value in storage is not what the program expects, then the location is not modified; instead, the current value of the location is loaded into a general register, in preparation for the program to loop back and try again. During the execution of COMPARE AND SWAP, no other CPU can perform a store access or interlocked-update access at the specified storage location.

To ensure successful updating of a common storage field by two or more CPUs, all updates must be done by means of an interlocked-update reference. See the programming notes of COMPARE AND SWAP for an example of how COMPARE AND SWAP can be unsuccessful due to an OR IMMEDIATE instruction executed by another CPU.

Setting a Single Bit

The following instruction sequence shows how the COMPARE AND SWAP instruction can be used to set a single bit in storage to one. Assume that the first byte of a word in storage called “WORD” contains eight flag bits.

```
LA 6,X'80' Put bit to be ORed into GR6
SLL 6,24 Shift left 24 places to align the byte to be ORed with the location of the flag bits within WORD
L 7,WORD Fetch current flag values
RETRY LR 8,7 Load flags into GR8
OR 8,6 Set bit to one
CS 7,8,WORD Store new flags if current flags unchanged, or re-fetch current flag values if changed
BC 4,RETRY If new flags are not stored, try again
```

The format of the COMPARE AND SWAP instruction is:

Machine Format

```
Op Code R/basesuperone R/basesuperthree S/basesupertwo
├───┬────┬────┬────┐
│ BA │ 7 │ 8 │/c5197/c5197/c5197/c5197│
└───┴────┴────┴────┘
```

Assembler Format

```
CS 7,8,WORD
```

The COMPARE AND SWAP instruction compares the first operand (general register 7 containing the current flag values) to the second operand in storage (WORD) while no CPU other than the one executing the COMPARE AND SWAP instruction is permitted to perform a store access or interlocked-update access at the specified storage location.

If the comparison is successful, indicating that the flag bits have not been changed since they were fetched, the modified copy in general register 8 is stored into WORD. If the flags have been changed, the compare will not be successful, and their new values are loaded into general register 7.

The conditional branch (BC) instruction tests the condition code and reexecutes the flag-modifying instructions if the COMPARE AND SWAP instruction indicated an unsuccessful comparison (condi-
tion code 1). When the COMPARE AND SWAP instruction is successful (condition code 0), the flags contain valid data, and the program exits from the loop.

The branch to RETRY will be taken only if some other program modifies the contents of WORD. This type of a loop differs from the typical “bit-spin” loop. In a bit-spin loop, the program continues to loop until the bit changes. In this example, the program continues to loop only if the value does change during each iteration. If a number of CPUs simultaneously attempt to modify a single location by using the sample instruction sequence, one CPU will fall through on the first try, another will loop once, and so on until all CPUs have succeeded.

**Updating Counters**

In this example, a 32-bit counter is updated by a program using the COMPARE AND SWAP instruction to ensure that the counter will be correctly updated. The original value of the counter is obtained by loading the word containing the counter into general register 7. This value is moved into general register 8 to provide a modifiable copy, and general register 6 (containing an increment to the counter) is added to the modifiable copy to provide the updated counter value. The COMPARE AND SWAP instruction is used to ensure valid storing of the counter.

The program updating the counter checks the result by examining the condition code. The condition code 0 indicates a successful update, and the program can proceed. If the counter had been changed between the time that the program loaded its original value and the time that it executed the COMPARE AND SWAP instruction, the execution would have loaded the new counter value into general register 7 and set the condition code to 1, indicating an unsuccessful update. The program must then repeat the update sequence until the execution of the COMPARE AND SWAP instruction results in a successful update.

The following instruction sequence performs the above procedure:

```
LA 6,1   Put increment (1) into GR6
L 7,CNTR Put original counter value into GR7
LOOP LR 8,7 Set up copy in GR8 to modify AR 8,6 Increment copy
CS 7,8,CNTR Update counter in storage BC 4,LOOP If original value had changed, update new value
```

The following shows two CPUs, A and B, executing this instruction sequence simultaneously: both CPUs attempt to add one to CNTR.

<table>
<thead>
<tr>
<th>CPU A</th>
<th>CPU B</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR7</td>
<td>GR8</td>
<td>CNTR</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>CPU A loads GR7 and GR8 from CNTR</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>CPU B loads GR7 and GR8 from CNTR</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>CPU B adds one to GR8</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>CPU A adds one to GR8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU A executes CS; successful match, store</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>CPU B executes CS; no match, GR7 changed to CNTR value</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>CPU B loads GR8 from GR7, adds one to GR8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU B executes CS; successful match, store</td>
</tr>
</tbody>
</table>

**Bypassing Post and Wait**

**Bypass Post Routine**

The following routine allows the SVC “POST” as used in MVS/ESA to be bypassed whenever the corresponding WAIT has not yet been executed, provided that the supervisor WAIT and POST routines use COMPARE AND SWAP to manipulate event control blocks (ECBs).

**Initial Conditions:**

- GR0 contains the POST code.
- GR1 contains the address of the ECB.
- GR5 contains 40 00 00 00

Appendix A. Number Representation and Instruction-Use Examples A-45
HSPOST OR 0,5 Set bit 1 of GR0 to one
L 3,0(1) GR3 = contents of ECB
LTR 3,3 ECB marked 'waiting'?
BC 4,PSVC Yes, execute post SVC
CS 3,0,0(1) No, store post code
BC 8,EXITHP Continue
PSVC POST (1),(0) ECB address is in GR1, post code in GR0
EXITHP [Any instruction]

The following routine may be used in place of the previous HSPOST routine if it is assumed that bit 1 of the contents of GR0 is already set to one and if the ECB is assumed to contain zeros when it is not marked “WAITING.”

HSPOST SR 3,3
CS 3,0,0(1)
BC 8,EXITHP
POST (1),(0)
EXITHP [Any instruction]

Bypass Wait Routine
A BYPASS WAIT function, corresponding to the BYPASS POST, does not use the CS instruction, but the FIFO LOCK/UNLOCK routines which follow assume its use.

HSWAIT TM 0(1),X'40'
BC 1,EXITHW If bit 1 is one, then ECB is already posted; branch to exit
WAIT ECB=(1)
EXITHW [Any instruction]

Lock/Unlock

When a common storage area larger than a doubleword is to be updated, it is usually necessary to provide special interlocks to ensure that a single program at a time updates the common area. Such an area is called a serially reusable resource (SRR).

In general, updating a list, or even scanning a list, cannot be safely accomplished without first “freezing” the list. However, the COMPARE AND SWAP and COMPARE DOUBLE AND SWAP instructions can be used in certain restricted situations to perform queuing and list manipulation. Of prime importance is the capability to perform the lock/unlock functions and to provide sufficient queuing to resolve contentions, either in a LIFO or FIFO manner. The lock/unlock functions can then be used as the interlock mechanism for updating an SRR of any complexity.

The lock/unlock functions are based on the use of a “header” associated with the SRR. The header is the common starting point for determining the states of the SRR, either free or in use, and also is used for queuing requests when contentions occur. Contentions are resolved using WAIT and POST. The general programming technique requires that the program that encounters a “locked” SRR must “leave a mark on the wall” indicating the address of an ECB on which it will WAIT. The “unlocking” program sees the mark and posts the ECB, thus permitting the waiting program to continue. In the two examples given, all programs using a particular SRR must use either the LIFO queuing scheme or the FIFO scheme; the two cannot be mixed. When more complex queuing is required, it is suggested that the queue for the SRR be locked using one of the two methods shown.

Lock/Unlock with LIFO Queuing for Contentions

The header consists of a word, that is, a four-byte field aligned on a word boundary. The word can contain zero, a positive value, or a negative value.

- A zero value indicates that the serially reusable resource (SRR) is free.
- A negative value indicates that the SRR is in use but no additional programs are waiting for the SRR.
- A positive value indicates that the SRR is in use and that one or more additional programs are waiting for the SRR. Each waiting program is identified by an element in a chained list. The positive value in the header is the address of the element most recently added to the list.

Each element consists of two words. The first word is used as an ECB; the second word is used as a pointer to the next element in the list. A negative value in a pointer indicates that the element is the last element in the list. The element is required only if the program finds the SRR locked and desires to be placed in the list.

The following chart describes the action taken for LIFO LOCK and LIFO UNLOCK routines. The routines following the chart allow enabled code to perform the actions described in the chart.
### LIFO LOCK Routine:

**Initial Conditions:**

- GR1 contains the address of the incoming element.
- GR2 contains the address of the header.

**LLOCK**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR 3,3</td>
<td>GR3 = 0</td>
</tr>
<tr>
<td>ST 3,0(1)</td>
<td>Initialize the ECB</td>
</tr>
<tr>
<td>LNR 0,1</td>
<td>GR0 = a negative value</td>
</tr>
<tr>
<td>CS 3,0,0(2)</td>
<td>Set the header to a negative value if the header contains zeros</td>
</tr>
<tr>
<td>BC 8,USE</td>
<td>Did the header contain zeros?</td>
</tr>
<tr>
<td>ST 3,4(1)</td>
<td>No, store the value of the header into the pointer in the incoming element</td>
</tr>
<tr>
<td>CS 3,1,0(2)</td>
<td>Store the address of the incoming element into the header</td>
</tr>
<tr>
<td>LA 3,0(0)</td>
<td>GR3 = 0</td>
</tr>
<tr>
<td>BC 7,TRYAGN</td>
<td>Did the header get updated?</td>
</tr>
<tr>
<td>WAIT ECB=(1)</td>
<td>Yes, wait for the resource; the ECB is in the incoming element</td>
</tr>
</tbody>
</table>

**USE**

[Any instruction]

### LIFO UNLOCK Routine:

**Initial Conditions:**

- GR2 contains the address of the header.

**LUNLK**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUNLK L 1,0(2)</td>
<td>GR1 = the contents of the header</td>
</tr>
<tr>
<td>A LTR 1,1</td>
<td>Does the header contain a zero?</td>
</tr>
<tr>
<td>BC 4,B</td>
<td>negative value?</td>
</tr>
<tr>
<td>L 0,4(1)</td>
<td>No, load the pointer from location A+4</td>
</tr>
<tr>
<td>CS 1,0,0(2)</td>
<td>the &quot;last in&quot; element and store it in the header</td>
</tr>
<tr>
<td>BC 7,A</td>
<td>Did the header get updated?</td>
</tr>
<tr>
<td>POST (1)</td>
<td>Yes, post the &quot;last in&quot; element</td>
</tr>
<tr>
<td>BC 15,EXIT</td>
<td>Continue</td>
</tr>
<tr>
<td>B SR 0,0</td>
<td>The header contains a negative value; free the SRR</td>
</tr>
<tr>
<td>CS 1,0,0(2)</td>
<td>element</td>
</tr>
<tr>
<td>BC 7,A</td>
<td>header and continue</td>
</tr>
</tbody>
</table>

**EXIT**

[Any instruction]

Note that the LOAD instruction L 1,0(2) at location LUNLK would have to be CS 1,1,0(2) if it were not for the rule concerning storage-operand consistency. This rule requires the LOAD instruction to fetch a four-byte operand aligned on a word boundary such that, if another CPU changes the word being fetched by an operation which is also at least word-consistent, either the entire new or the entire old value of the word is obtained, and not a combination of the two. (See "Storage-Operand Consistency" on page 5-98.)

### Lock/Unlock with FIFO Queuing for Contentions

The header always contains the address of the most recently entered element. The header is originally initialized to contain the address of a posted ECB. Each program using the serially reusable resource (SRR) must provide an element regardless of whether contention occurs. Each program then enters the address of the element which it has provided into the header, while simultaneously it removes the address previously contained in the header. Thus, associated with any particular program attempting to use the SRR are two elements, called the "entered element" and the "removed element." The "entered element" of one program becomes the "removed element" for the immediately following program. Each program then waits on the removed element, uses the SRR, and then posts the entered element.

When no contention occurs, that is, when the second program does not attempt to use the SRR until after the first program is finished, then the POST of the first program occurs before the WAIT of the second program. In this case, the bypass-post and bypass-wait routines described in the preceding section are applicable. For simplicity,
these two routines are shown only by name rather than as individual instructions.

In the example, the element need be only a single word, that is, an ECB. However, in actual practice, the element could be made larger to include a pointer to the previous element, along with a program identification. Such information would be useful in an error situation to permit starting with the header and chaining through the list of elements to find the program currently holding the SRR.

It should be noted that the element provided by the program remains pointed to by the header until the next program attempts to lock. Thus, in general, the entered element cannot be reused by the program. However, the removed element is available, so each program gives up one element and gains a new one. It is expected that the element removed by a particular program during one use of the SRR would then be used by that program as the entry element for the next request to the SRR.

It should be noted that, since the elements are exchanged from one program to the next, the elements cannot be allocated from storage that would be freed and reused when the program ends. It is expected that a program would obtain its first element and release its last element by means of the routines described in “Free-Pool Manipulation.”

The following chart describes the action taken for FIFO LOCK and FIFO UNLOCK.

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO LOCK</td>
<td>Store address A into the header. WAIT; the ECB is at the location addressed by the old contents of the header.</td>
</tr>
<tr>
<td>(the incoming element is at location A)</td>
<td></td>
</tr>
<tr>
<td>FIFO UNLOCK</td>
<td>POST; the ECB is at location A.</td>
</tr>
</tbody>
</table>

The following routines allow enabled code to perform the actions described in the previous chart.

**FIFO Lock Routine:**

Initial conditions:

- GR3 contains the address of the header.
- GR4 contains the address, A, of the element currently owned by this program. This element becomes the entered element.

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOCK</td>
<td>GR2 now contains address of element to be entered</td>
</tr>
<tr>
<td>LR 2,4</td>
<td>SR 1,1</td>
</tr>
<tr>
<td>SR 1,1</td>
<td>GR1 = 0</td>
</tr>
<tr>
<td>ST 1,0(2)</td>
<td>ST 1,0(2) Initialize the ECB</td>
</tr>
<tr>
<td>L 1,0(3)</td>
<td>L 1,0(3) GR1 = contents of the header</td>
</tr>
<tr>
<td>TRYAGN</td>
<td>CS 1,2,0(3) Enter address A into header while remembering old contents of header into GR1; GR1 now contains address of removed element</td>
</tr>
<tr>
<td>BC 7,TRYAGN</td>
<td>LR 4,1 Removed element becomes new currently owned element</td>
</tr>
<tr>
<td>HSWAIT</td>
<td>Perform bypass-wait routine; if ECB already posted, continue; if not, wait; GR1 contains the address of the ECB</td>
</tr>
</tbody>
</table>

**FIFO Unlock Routine:**

Initial conditions:

- GR2 contains the address of the removed element, obtained during the FLOCK routine.
- GR5 contains 40 00 00 00

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNLK</td>
<td>LR 1,2 Place address of entered element in GR1; GR1 = address of ECB to be posted</td>
</tr>
<tr>
<td>SR 0,0</td>
<td>SR 0,0 GR0 = 0; GR0 has a post code of zero</td>
</tr>
<tr>
<td>OR 0,5</td>
<td>OR 0,5 Set bit 1 of GR0 to one</td>
</tr>
<tr>
<td>HSPOST</td>
<td>HSPOST Perform bypass-post routine; if ECB has not been waited on, then mark posted and continue; if it has been waited on, then post</td>
</tr>
</tbody>
</table>

**Free-Pool Manipulation**

It is anticipated that a program will need to add and delete items from a free list without using the lock/unlock routines. This is especially likely since the lock/unlock routines require storage elements for queuing and may require working storage. The lock/unlock routines discussed previously allow simultaneous lock routines but permit only one unlock routine at a time. In such a situation, multiple additions and a single deletion to the list...
may all occur simultaneously, but multiple deletions cannot occur at the same time. In the case of a chain of pointers containing free storage buffers, multiple deletions along with additions can occur simultaneously. In this case, the removal cannot be done using the COMPARE AND SWAP instruction without a certain degree of exposure.

Consider a chained list of the type used in the LIFO lock/unlock example. Assume that the first two elements are at locations A and B, respectively. If one program attempted to remove the first element and was interrupted between the fourth and fifth instructions of the LUNLK routine, the list could be changed so that elements A and C are the first two elements when the interrupted program resumes execution. The COMPARE AND SWAP instruction would then succeed in storing the value B into the header, thereby destroying the list.

The probability of the occurrence of such list destruction can be reduced to near zero by appending to the header a counter that indicates the number of times elements have been added to the list. The use of a 32-bit counter guarantees that the list will not be destroyed unless the following events occur, in the exact sequence:

1. An unlock routine is interrupted between the fetch of the pointer from the first element and the update of the header.

2. The list is manipulated, including the deletion of the element referenced in 1, and exactly \(2^{32}\) (or an integer multiple of \(2^{32}\)) additions to the list are performed. Note that this takes on the order of days to perform in any practical situation.

3. The element referenced in 1 is added to the list.

4. The unlock routine interrupted in 1 resumes execution.

The following routines use such a counter in order to allow multiple, simultaneous additions and removals at the head of a chain of pointers.

The list consists of a doubleword header and a chain of elements. The first word of the header contains a pointer to the next element in the list. A zero value indicates the end of the list.

The following chart describes the free-pool-list manipulation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD TO LIST (the incoming element is at location A)</td>
<td>Store the first word of the header into location A. Store the address A into the first word of the header. Decrement the second word of the header by one.</td>
</tr>
<tr>
<td>DELETE FROM LIST</td>
<td>The list is empty. Set the first word of the header to the value of the contents of location A. Use element A.</td>
</tr>
</tbody>
</table>

The following routines allow enabled code to perform the free-pool-list manipulation described in the above chart.

**ADD TO FREE LIST Routine:**

Initial Conditions:

GR2 contains the address of the element to be added.

GR4 contains the address of the header.

ADDQ 0,1,0(4) GR0,GR1 = contents of the header

TRYAGN ST 0,0(2) Point the new element to the top of the list

LR 3,1 Move the count to GR3

BCTR 3,0 Decrement the count

CDS 0,2,0(4) Update the header

BC 7,TRYAGN

**DELETE FROM FREE LIST Routine:**

Initial conditions:

GR4 contains the address of the header.

DELETQ 2,3,0(4) GR2,GR3 = contents of the header

TRYAGN LTR 2,2 Is the list empty?

BC 8,EMPTY Yes, get help

L 0,0(2) No, GR0 = the pointer from the first element

LR 1,3 Move the count to GR1

CDS 2,0,0(4) Update the header

BC 7,TRYAGN

USE [Any instruction] The address of the removed element is in GR2

Note that the LM (LOAD MULTIPLE) instructions at locations ADDQ and DELETQ would have to be CDS (COMPARE DOUBLE AND SWAP)
instructions if it were not for the rule concerning storage-operand consistency. This rule requires the LOAD MULTIPLE instructions to fetch an eight-byte operand aligned on a doubleword boundary such that, if another CPU changes the doubleword being fetched by an operation which is also at least doubleword-consistent, either the entire new or the entire old value of the doubleword is obtained, and not a combination of the two. (See "Storage-Operand Consistency" on page 5-98.)

PERFORM LOCKED OPERATION (PLO)

The PERFORM LOCKED OPERATION instruction can be used in a multiprogramming or multiprocessing environment to perform compare, load, compare-and-swap, and store operations on two or more discontiguous locations that can be words or doublewords. The operations are performed as an atomic set of operations under the control of a lock that is held only for the duration of the execution of a single PERFORM LOCKED OPERATION instruction, as opposed to across the execution of multiple instructions. Since lock contention is resolved by the CPU and is very brief, the program need not include a method for dealing with the case when the lock to be used is held by a program being executed by another CPU. Also, there need be no concern that the program may be interrupted while it holds a lock, since PERFORM LOCKED OPERATION will complete its operation and release its lock before an interruption can occur.

PERFORM LOCKED OPERATION can be thought of as performing concurrent interlocked updates of multiple operands. However, the instruction does not actually perform any interlocked update, and a serially reusable resource cannot be updated predictably through the use of both PERFORM LOCKED OPERATION and conditional-swapping instructions (CS and CDS).

Following is an example of how PERFORM LOCKED OPERATION can be used to add an element at the beginning of a queue.

Assume the following variables associated with the queue: S, which is a sequence number that is incremented anytime the queue is changed; H (for head), which is the address of the first element on the queue; and C, which is a count of the number of elements on the queue. Assume a queue element contains a variable, F (for forward), which is the address of the next element on the queue. If a new element, N, is to be enqueued at the head of the queue, that can be done by setting F in N to H and then performing the following atomic set of operations:

\[
\begin{align*}
S + 1 & \rightarrow S \\
A(N) & \rightarrow H \\
C + 1 & \rightarrow C
\end{align*}
\]

where A(N) is the address of N.

The enqueueing of N can be done by means of the following steps:

1. Obtain consistent values of S, H, and C, meaning obtain S and obtain the H and C that are consistent with that value of S.
2. Store H in N.F.
3. By means of PLO.csdst (PERFORM LOCKED OPERATION performing compare and swap and double store), with S as the swap variable and H and C as the store variables, add one to S, set H to A(N), and add one to C, provided that S still has the value obtained in step 1. If S has already been changed, go back to step 1.

Consistent values of S, H, and C cannot necessarily be obtained simply by using three LOAD instructions because a PERFORM LOCKED OPERATION instruction being executed by another CPU may have completed an update of S but not yet of H or C. In this case, the three LOAD instructions will obtain the new S but the old H or C. However, as will be described, it may be possible to use three LOAD instructions.

If S is obtained while holding the lock, meaning by means of PERFORM LOCKED OPERATION, then H and C can be obtained by LOAD instructions since no other CPU can subsequently change H or C without changing S, as observed when the lock is held.

The parameter list used by the PLO.csdst is as follows, assuming the access-register mode is not used:
The program is as follows:

```
LA RT,H  Initialize addresses in PL
         (T = temp)
ST RT,PL+76 Op4 address (address of H)
LA RT,C  
ST RT,PL+108 Op6 address (address of C)
LA RN,N  Address of N
ST RN,PL+60 Initialize op3 in PL
         (address of N)
LA R1,S  PLT address = address of S
------------------------------------------------
SR RS,RS Dummy S. CC1 will probably be set
SR R0,R0 Function code 0 (compare and load)
PLO RS,S,RS,S Obtain S while holding lock
------------------------------------------------
LA R0,16 Function code 16 (csdst)
LOOP L RT,H Consistent H
ST RT,OFSTF(RN) OFSTF = offset of F in N
L RT,C Consistent C
LA RT,1(RT) C+1
ST RT,PL+92 Initialize op5 in PL (C+1)
LA RSP,1(RS) RS/RSP = even/odd pair.
         S+1 in RSP
PLO RS,S,0,PL
BNZ LOOP Br if S changed (if CC not 0)
```

Note the following about the first PERFORM LOCKED OPERATION instruction (PLO.cl). If S is not zero (which is probably true), S (the second operand, op2) is loaded into RS (the first-operand comparison value, op1c). If S is zero, S (the fourth operand, op4) is loaded into RS (the third operand, op3). Either of these loads occurs while the lock is held. It is unnecessary to test the condition code to determine which load occurred.

The above program may be a simplification. If the queue has associated with it a variable, T (for tail), that is the address of the last element on the queue, and the queue is currently empty, T also must be set when N is added to the queue. This would require a different program using a compare-and-swap-and-triple-store operation.

If the queue is added to, deleted from, and rearranged by means of PERFORM LOCKED OPERATION instructions in which the sequence number, S, is always the second operand, then, since the definition of PERFORM LOCKED OPERATION specifies that the second operand is always stored last, the first PERFORM LOCKED OPERATION instruction in the above program can be replaced by a LOAD instruction. The three instructions within the dashed lines would be replaced by L RS,S.

### Sorting Instructions

#### Tree Format

Two instructions, **COMPARE AND FORM CODEWORD** and **UPDATE TREE**, refer to a tree — a data structure with a specific format. A tree consists of some number (always odd) of consecutively numbered nodes. Node 1 is the root of the tree. Every node except the root has one parent node in the same tree. Every parent node has two son nodes. Every even-numbered node is the left son of its parent node, and every odd-numbered node (except node 1) is the right son of its parent node. Division by two (ignoring remainder) of the node number gives the parent node number. Nodes with sons are also called internal nodes, and nodes without sons are called terminal nodes. Figure A-5 on page A-53 illustrates schematically a 21-node tree with arrows drawn from each parent node to each son node.

A tree is used for merging several sorted sequences of records into a single merged sequence of records. At each step in the merging process, there exists the initial part of the merged sequence and the remaining parts of each of the sorted sequences that are being merged. Each
step consists in selecting the lowest record (the record with the lowest key when sorting in ascending sequence) from all of the as yet unmerged parts of the sorted sequences and adding it to the merged sequence. Each terminal node in the tree represents one of the sorted sequences. The number of internal nodes in the tree is one less than the number of sorted sequences. Each internal node conceptually contains one record from each of the sorted sequences but one; these are the lowest records, from all but one of the sorted sequences, that have not yet been added to the merged sequence. In addition, there is the lowest record from the one remaining sorted sequence. This additional record is compared and interchanged with nodes of the tree to select the record to be added next to the merged sequence. This processing begins with the parent of the terminal node that represents the one remaining sorted sequence, and it continues from that node along the path to the root of the tree. The selected record emerges from the root of the tree.

The tree may perhaps be most easily explained by considering each node to represent a comparison operation in an “elimination tournament” to find the lowest record. After the tournament has been completed, each node has an associated “loser” record which had a higher key in the comparison represented by that node. Besides a loser record at each node, there is one record (the “winner”) which is not associated with any node since it never compared high. The next step would be to introduce a new record from the same sorted sequence from which the winner record originated and replay the tournament with the new record in place of the former winner. It can be seen that it is unnecessary to do all the comparisons represented by all the nodes in the tree — most of them are unaffected by the new record replacing the former winner. In fact, it is sufficient to redo only those node comparisons in which the former winner record participated. Each new record is inserted into the tree at the terminal node that represents the sorted sequence containing the record. The use of the tree assumes that programming provides a method of remembering at which terminal node each winning record originated. The instruction UPDATE TREE allows for a new record to be inserted at a terminal node and the tree to be updated so that a new winner record is left in the general registers.

Rather than comparing the actual keys of records, much of the merge logic can be performed using “codewords” to represent a record key rather than referring to actual keys. The value of a codeword at a node in the tree depends not only on the record's key but also on the key of the winning record in the last comparison at that node. The codeword consists of two parts:

1. Bits 16-31 contain the one’s complement of the first halfword in which the record key differs from that of the node's winning record.
2. Bits 0-15 specify the byte offset of the halfword in this record's key just beyond the halfword value (complemented) in bit positions 16-31.

When comparing records in the path of the last winner record, if the new record is also represented by a codeword resulting from a comparison with the last winner, all codewords in the update path are with respect to the same winner. When comparing such codewords, a high codeword represents a low key and vice versa. Thus, when codewords are unequal, a node entry with a high codeword (representing a low actual key) should move up the tree.

In the case of a tie value of codewords, it is necessary to refer to the actual keys. This is done by the instruction COMPARE AND FORM CODEWORD, which resolves the ambiguity and computes a new codeword for the high-key (loser) record.

The eight bytes at each node of a tree consist of (1) a codeword for this record, computed with respect to the last record which compared low against this record and (2) a parameter usable to locate this record, for example, a direct or indirect address.

The instruction UPDATE TREE is so defined that tree updating stops after equal codewords are detected and the tie-breaking instruction COMPARE AND FORM CODEWORD can be used, after which UPDATE TREE can resume tree updating at the point where equal codewords were previously found.

COMPARE AND FORM CODEWORD may alternatively be used for merging in descending sequence. In that case, bits 16-31 of the codeword at a node contain the true value of the first halfword in which the record key differs from
that of the node’s winning record. When the descending option of COMPARE AND FORM CODEWORD is used, the higher of two codewords represents the higher key.

Figure A-5. Schematic Diagram of Merge Control Tree with 21 Nodes

Example of Use of Sort Instructions

An example illustrates how the instructions UPDATE TREE and COMPARE AND FORM CODEWORD may be used in the merge operation within a sort program. A five-way merge requires a tree data structure with four internal nodes and five terminal-node positions. The schematic diagram shown later in this section illustrates such a tree, containing four internal nodes (not counting the dummy node) and five input sequences for a merge, one sequence at each terminal-node position. Each record in an input sequence in the diagram is indicated by its address. The actual record contents are shown in Figure A-7 on page A-57. Each record contains 16 bytes, consisting of the following fields:

<table>
<thead>
<tr>
<th>Byte Offset (hexadecimal)</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>Six-byte record key.</td>
</tr>
<tr>
<td>6-7</td>
<td>Halfword node index specifying the input sequence of the next record of this input sequence.</td>
</tr>
<tr>
<td>8-B</td>
<td>Address of the next record in the same input sequence.</td>
</tr>
<tr>
<td>C-F</td>
<td>This chaining field is initially zero. At the completion of the merge, this field is to contain the address of the next record in the merged sequence.</td>
</tr>
</tbody>
</table>

The merge process forms a single sorted sequence from five input sequences, each of which is in sorted order. This process can be subdivided into three steps:

1. A priming step takes the first record from each of the five input sequences and places them in the tree data structure. For each record to be introduced into the tree, first its codeword value is computed with respect to the lowest possible key value of all zeros. This codeword, with a second word which contains the address of the actual record, forms a doubleword node value that can be placed at the appropriate node. After priming, the node values, one each from each of the five input sequences, will have been placed in the tree so that each of the four internal nodes contains one node value and the node value for a winner record has emerged from the root of the tree.

2. After each winner emerges from the tree, the main merge process is performed repeatedly. Each iteration introduces the node value for one new record into the tree and produces a node value for a new winner record. The tree plus the winner must at all times contain precisely one node value from each input sequence being merged. Therefore, the new node value that is introduced into the tree on each iteration must come from the same input sequence from which the winner node value in the preceding iteration originated.

3. When the node value for the last record of an input sequence emerges as a winner, there is no successor record from that input sequence to be introduced into the tree on the next iteration. Hence, the order of the merge must be reduced by one for each such occurrence.
This runout process will consist of one or more iterations for each of a four-way, three-way, two-way, and one-way merge. The onset of runout occurs in the example when it is found that the next input record from a sequence is lower than its predecessor (a sequence break).

The priming process is discussed next, and the state of the tree is shown after priming is complete. Then, a short program that uses the instructions UPDATE TREE and COMPARE AND FORM CODEWORD to perform the main merge is described. An abbreviated trace is then presented to show the status of the tree and certain general registers for 16 iterations of the main merge. The runout process is not discussed in this example.

Priming begins by forming the node value for the first record of each input sequence. The first word of the node value is the codeword formed by executing COMPARE AND FORM CODEWORD on a record key containing all binary zeros. The second word of the node value is the address of the record represented by that node value. The node values for the first record of each input sequence are:

<table>
<thead>
<tr>
<th>Sequence Index</th>
<th>Node Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>0006 FFFC 0000 1030</td>
</tr>
<tr>
<td>30</td>
<td>0006 FFFB 0000 1040</td>
</tr>
<tr>
<td>38</td>
<td>0006 FFFA 0000 1050</td>
</tr>
<tr>
<td>40</td>
<td>0004 FFFE 0000 1080</td>
</tr>
<tr>
<td>48</td>
<td>0006 FF00 0000 1060</td>
</tr>
</tbody>
</table>

In the example, the tree data structure is assumed to have base address X'1000', which is kept in general register 4 (to match the expected use in UPDATE TREE). Similarly, internal-node index values and input-sequence index values are always used from general register 5.

Although the tree-priming program is not part of this example, the UPDATE TREE instruction is used in creating it as follows. First, the codeword position for each internal node of the tree is initialized to all ones (X'FFFF FFFF'). This artifice fills the tree with dummy low records. Then, for each record in the table, (1) the sequence index is loaded into general register 5, (2) the node value is loaded into general registers 0 and 1, and (3) UPDATE TREE is executed. At the completion of this priming process, the tree-node contents in the example are as shown on line 0 of Figure A-9 on page A-59. The contents of the general registers are as shown on the first line of Figure A-8 on page A-58.

The figure illustrating the program for the main merge is divided into three groups of columns, containing the absolute program, the general-register trace, and the symbolic program. The first part of the program extends from symbolic locations L1 through L2; it introduces a new record into the tree and executes an UPDATE TREE instruction. If no tied codewords are encountered in UPDATE TREE, then the BRANCH ON CONDITION instruction following UPDATE TREE loops back to L1 to introduce the next record into the tree. This BRANCH ON CONDITION instruction is suitable for use when UPDATE TREE operates in accordance with either its method 1 (setting condition code 1) or its method 2 (setting condition code 3). (The preceding sentence applies to 370-XA. In ESA/370 and ESA/390, UPDATE TREE operates in accordance with only method 2, which is not to say that it cannot set condition code 1. Method 2, but not method 1, tests for the condition that sets condition code 3.)

If UPDATE TREE encounters tied codewords, then the UPDATE TREE instruction is completed, the subsequent BRANCH ON CONDITION instruction does not branch, and control falls through to the second part of the program, which handles entries with tied codewords. This part then branches back to UPDATE TREE at L2, which resumes the tree updating. It is possible for tied codewords to be encountered at any level in the tree (or indeed at all levels), so that the tied-codeword part of the program may be entered up to three times for each record introduced.

The general-register trace for the first part of the main merge shows the contents of the first seven general registers after each instruction is executed during the first iteration. Note that the merged-chain field (at 1140) serves as the anchor for the merged-chain address chain through the records. The trace shows only the lower half of certain general registers, whose upper half is always zero.

Figure A-9 on page A-59 gives an abbreviated trace of the entire main merge of 16 records. For each record introduced into the tree, there are one or more lines (always an odd number) given in the figure to show the tree updating, which results
finally in a winner in GR0 and GR1. The first line for each record shows the values of GR5, GR2, and GR3 before the first or only execution of UPDATE TREE. For the even-numbered lines, the storage updating by UPDATE TREE of tree nodes is shown (read left to right to follow the order of swapping). For example, consider line 10 and the corresponding UPDATE TREE: since GR5 contains 28, the first storage node examined is 1010 (refer to the schematic diagram). Since the codeword in GR0 is 0004 FFFE (same as for GR2), which is less than that of the word at 1010 (0006 FFF0), the doubleword at 1010 is swapped with that in GR0 and GR1. A second comparison at 1008 in the same execution of UPDATE TREE causes another register-storage doubleword swap, which leaves the winner (record 1040) in GR1 at the completion of UPDATE TREE (see the column at the far right of Figure A-9 on page A-59).

When a codeword comparison is made which does not result in a tie or a swap (that is, when the storage-codeword value is low), an asterisk appears in the trace for that storage entry.

When equal codewords are found, the execution of UPDATE TREE is completed. The following line in each such case shows the result of the tied-codeword routine, which always stores a new codeword and may also store a new record address before branching back to L2 to execute UPDATE TREE again. In this line, the notation “loses” or “wins” means that the node loses or wins, respectively.

The tie-break trace part of Figure A-8 on page A-58 shows the treatment of the third record (that is, the first record for which UPDATE TREE encounters a tied codeword). This corresponds to line 31 in Figure A-9 on page A-59.

The following is a summary of the steps that are needed to use this example for verification purposes:

1. Initialize storage as follows:
   a. 1008 through 102F from line 0 of Figure A-9 on page A-59
   b. 1030 through 114F from Figure A-7 on page A-57
   c. 1150 through 1189 from Figure A-8 on page A-58

2. Initialize GRs per first line in Figure A-8 and trace first record per Figure A-8.

3. Trace to completion of each UPT or BC 15,L2 (once for each line of Figure A-9). A detailed trace of the GRs for the tied-codeword part of line 31 of Figure A-9 is given in the lower part of Figure A-8.

4. Verify that addresses in the chain beginning at 103C and continuing through 114C are as shown in the right-hand column of Figure A-7.
Note: Each node and input sequence is identified by a number which is the hexadecimal node index. Each input sequence is given as a list of record addresses (also in hexadecimal).

Figure A-6. Schematic Diagram for Example of Merge to Be Performed
Figure A-7. Contents of Records to Be Merged
<table>
<thead>
<tr>
<th>Absolute</th>
<th>General-Register Trace</th>
<th>Symbolic Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc</td>
<td>INSTR</td>
<td>GR0</td>
</tr>
<tr>
<td>1150</td>
<td>5010600C</td>
<td>0000</td>
</tr>
<tr>
<td>1154</td>
<td>48501006</td>
<td></td>
</tr>
<tr>
<td>1158</td>
<td>58301008</td>
<td></td>
</tr>
<tr>
<td>115C</td>
<td>1861</td>
<td></td>
</tr>
<tr>
<td>1153</td>
<td>1B22</td>
<td></td>
</tr>
<tr>
<td>1160</td>
<td>B21A0004</td>
<td></td>
</tr>
<tr>
<td>1164</td>
<td>4720418A</td>
<td></td>
</tr>
<tr>
<td>1168</td>
<td>1813</td>
<td></td>
</tr>
<tr>
<td>116A</td>
<td>1802</td>
<td></td>
</tr>
<tr>
<td>116C</td>
<td>0102</td>
<td></td>
</tr>
<tr>
<td>116E</td>
<td>47504150</td>
<td></td>
</tr>
<tr>
<td>1172</td>
<td>88200010</td>
<td></td>
</tr>
<tr>
<td>1176</td>
<td>B21A0004</td>
<td></td>
</tr>
<tr>
<td>1177</td>
<td>50254000</td>
<td></td>
</tr>
<tr>
<td>117E</td>
<td>47C0416C</td>
<td></td>
</tr>
<tr>
<td>1182</td>
<td>50354004</td>
<td></td>
</tr>
<tr>
<td>1186</td>
<td>47F0416C</td>
<td></td>
</tr>
<tr>
<td>118A</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

* Fall through on tied codewords
* GR values for tie-break trace

L1 ST 1,12,(6) Store merged-chain address
LH 5,6,(1) Load node index of input sequence of winner
L 3,8,(1) Load successor-record address
LR 6,1 Save old winner address for next merged-chain store
SR 2,2 Zero GR2 as initial offset
CFC 4 Compute codeword of new record based on last winner
BC 2,L3 Exit on CC=2 (sequence break)
LR 1,3 Move new record entry to GRs 0-1
L2 UPT Update tree data structure
BC 5,L1 If no codeword tie found, branch to next iteration

Figure A-8. Program for Main Merge
<table>
<thead>
<tr>
<th>L#</th>
<th>GR 5</th>
<th>GR2</th>
<th>GR3</th>
<th>Comment</th>
<th>Storage Trace of Node Entries</th>
<th>General Regs after UPT or BC 15_,L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0004FFFE 1080 0006FFFA 1050 0006FFFD 1060 0006FFFB 1040 0006FFFC 1030</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>28</td>
<td>0004FFFE</td>
<td>1070</td>
<td>No tie</td>
<td>0004FFFE 1070 0006FFFD 1060 0006FFFB 1040</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>00040000</td>
<td>1080</td>
<td>No tie</td>
<td>00040000 1080 * 0006FFFA 1050</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>00040000</td>
<td>1090</td>
<td>CC = 0 Loses No tie</td>
<td>Tie 0006FFFD 00040000 1090 0006FFFA 1050</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>00040000</td>
<td>1090</td>
<td>CC = 0 Equal No tie</td>
<td>00040000 1090 * 0006FFFA 1050</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>0002FF7F</td>
<td>10F0</td>
<td>No tie</td>
<td>0002FF7F 10F0 00040000 10A0 ** 0004FFFE 1080 0004FFFE 1080</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>28</td>
<td>0002FF7F</td>
<td>1000</td>
<td>CC = 0 Wins No comp</td>
<td>0002FF7F 1000 Tie 0006FFFA 10A0 00040000 10A0 00040000 10A0</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>30</td>
<td>0002FF7F</td>
<td>10E0</td>
<td>No tie</td>
<td>Tie 00001070 00040000 1090 0006FFFA 1050</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>81</td>
<td>0002FF7F</td>
<td>1100</td>
<td>CC = 0 Wins No tie</td>
<td>Tie 0006FFFAF 1100 0002FF7F 10F0 0002FF7F 1000</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>30</td>
<td>0000010C</td>
<td>10C0</td>
<td>No tie</td>
<td>** 000010C 10C0</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>30</td>
<td>00020000</td>
<td>1140</td>
<td>No tie</td>
<td>00020000 1140 * 0002FF7F 10E0 0002FFFE 1000</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>28</td>
<td>0002FF7F</td>
<td>1110</td>
<td>CC = 0 Wins CC = 0</td>
<td>Tie 0004FFFD 1110 0002FF7F 10F0 0002FF7F 10F0</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>30</td>
<td>00020000</td>
<td>1130</td>
<td>CC = 0 Loses No tie</td>
<td>Tie 00060000 00020000 1130 00020000 1130</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>40</td>
<td>0002FF6F</td>
<td>1120</td>
<td>No tie</td>
<td>0002FF6F 1120 * 0006FFFA 1100</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>48</td>
<td>00020000</td>
<td>1140</td>
<td>No tie</td>
<td>00020000 1140 * 0004FFFC 1110</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>28</td>
<td>00020000</td>
<td>1140</td>
<td>No tie</td>
<td>00020000 1140 * 0002FF6F 1120</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>40</td>
<td>00020000</td>
<td>1140</td>
<td>CC = 0 Equal CC = 0</td>
<td>Tie 80001140 00020000 1140 00020000 1140</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>38</td>
<td>00020000</td>
<td>1050</td>
<td>Branch</td>
<td>00020000 1140 00020000 1140</td>
<td></td>
</tr>
</tbody>
</table>

Figure A-9 (Part 1 of 2). Abbreviated Trace of Main Merge Processing
Explanation:

1 Line 0 shows the values in the tree after it is primed.

* Means no swap.

** Means no swap if UPDATE TREE method 1 is used or no examination if UPDATE TREE method 2 is used. Only method 2 is included in ESA/370 and ESA/390.

CC = 0 UPDATE TREE finds a tie and sets condition code 0.

Loses The tied-codeword routine finds that the node loses.

Wins The tied-codeword routine finds that the node wins.

Equal The tied-codeword routine finds that the keys are equal.

Branch Branches to terminate at 118A on sequence break.

No comp No compare.

Figure A-9 (Part 2 of 2). Abbreviated Trace of Main Merge Processing
Appendix B. Lists of Instructions

The following figures list instructions by name, mnemonic, and operation code. Some models may offer instructions that do not appear in the figures, such as those provided for assists or as part of special or custom features.

The operation code for the interpretive execution facility is not included in this appendix. See the publication IBM System/370 Extended Architecture Interpretive Execution, SA22-7095, for the operation code associated with that facility.

The operation code 00 hex with a two-byte instruction format is allocated for use by the program when an indication of an invalid operation is required. It is improbable that this operation code will ever be assigned to an instruction implemented in the CPU.

Explanation of Symbols in “Characteristics” and “Page” Columns:

- \( \hat{c} \) Causes serialization and checkpoint synchronization.
- \( \hat{c}^1 \) Causes serialization and checkpoint synchronization when the \( M_1 \) and \( R_2 \) fields contain all ones and all zeros, respectively.
- \( \hat{c}^2 \) Causes serialization and checkpoint synchronization when the state entry to be unstacked is a program-call state entry.
- \$ Causes serialization.
- A Access exceptions for logical addresses.
- \( A^1 \) Access exceptions; not all access exceptions may occur; see instruction description for details.
- AI Access exceptions for instruction address.
- AS ASN-translation-specification and special-operation exceptions.
- AT ASN-translation-specification exception.
- B PER branch event.
- \( B_1 \) \( B_1 \) field designates an access register in the access-register mode.
- \( B_2 \) \( B_2 \) field designates an access register in the access-register mode.
- BP \( B_2 \) field designates an access register when PSW bits 16 and 17 have the value 01.
- C Condition code is set.
- Da AFP-register data exception.
- Db BFP-instruction data exception.
- Dd Decimal-operand data exception.
- DE DAT-enhancement facility.
- DF Decimal-enhancement facility.
- DK Decimal-divide exception.
- DM Depending on the model, DIAGNOSE may generate various program exceptions and may change the condition code.
- E E instruction format.
- E2 Extended-translation facility 2.
- EO HFP-exponent-overflow exception.
- ES Expanded-storage facility.
- EU HFP-exponent-underflow exception.
- EX Execute exception.
- FC Designation of access registers depends on the function code of the instruction.
- FK HFP-floating-point-divide exception.
- G0 Instruction execution includes the implied use of general register 0.
- G1 Instruction execution includes the implied use of general register 1.
- G2 Instruction execution includes the implied use of general register 2.
- G4 Instruction execution includes the implied use of general register 4.
- GM Instruction execution includes the implied use of multiple general registers.
- GS Instruction execution includes the implied use of general register 1 as the subsystem-identification word.
- HM HFP-multiply-add/subtract facility.
- IF Fixed-point-overflow exception.
- I Interruptible instruction.
- IK Fixed-point-divide exception.
- IS Interruptible instruction and special-operation exception.
- I1 Access register 1 is implicitly designated in the access-register mode.
- I4 Access register 4 is implicitly designated in the access-register mode.
- L New condition code is loaded.
- LD Long-displacement facility.
- LS HFP-significance exception.
- MD Designation of access registers in the access-register mode is model-dependent.
- MO Monitor event.
- MS Message-security assist.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Instruction is new in z/Architecture as compared to ESA/390.</td>
</tr>
<tr>
<td>N3</td>
<td>Instruction is new in z/Architecture and has been added to ESA/390.</td>
</tr>
<tr>
<td>OP</td>
<td>Operand exception.</td>
</tr>
<tr>
<td>P</td>
<td>Privileged-operation exception.</td>
</tr>
<tr>
<td>Q</td>
<td>Privileged-operation exception for semi-privileged instructions.</td>
</tr>
<tr>
<td>R1</td>
<td>( R_{1} ) field designates an access register in the access-register mode.</td>
</tr>
<tr>
<td>R2</td>
<td>( R_{2} ) field designates an access register in the access-register mode.</td>
</tr>
<tr>
<td>RI</td>
<td>RI instruction format.</td>
</tr>
<tr>
<td>RIE</td>
<td>RIE instruction format.</td>
</tr>
<tr>
<td>RIL</td>
<td>RIL instruction format.</td>
</tr>
<tr>
<td>RM</td>
<td>( R_{1} ) field designates an access register in the access-register mode, and access-register 1 also is used in the access-register mode.</td>
</tr>
<tr>
<td>RR</td>
<td>RR instruction format.</td>
</tr>
<tr>
<td>RRE</td>
<td>RRE instruction format.</td>
</tr>
<tr>
<td>RRF</td>
<td>RRF instruction format.</td>
</tr>
<tr>
<td>RS</td>
<td>RS instruction format.</td>
</tr>
<tr>
<td>RSL</td>
<td>RSL instruction format.</td>
</tr>
<tr>
<td>RSY</td>
<td>RSY instruction format.</td>
</tr>
<tr>
<td>RX</td>
<td>RX instruction format.</td>
</tr>
<tr>
<td>RXE</td>
<td>RXE instruction format.</td>
</tr>
<tr>
<td>RXF</td>
<td>RXF instruction format.</td>
</tr>
<tr>
<td>RXY</td>
<td>RXY instruction format.</td>
</tr>
<tr>
<td>S</td>
<td>S instruction format.</td>
</tr>
<tr>
<td>SE</td>
<td>Special operation, stack-empty, stack-specification, and stack-type exceptions.</td>
</tr>
<tr>
<td>SF</td>
<td>Special-operation, stack-full, and stack-specification exceptions.</td>
</tr>
<tr>
<td>SI</td>
<td>SI instruction format.</td>
</tr>
<tr>
<td>SIY</td>
<td>SIY instruction format.</td>
</tr>
<tr>
<td>SO</td>
<td>Special-operation exception.</td>
</tr>
<tr>
<td>SP</td>
<td>Specification exception.</td>
</tr>
<tr>
<td>SQ</td>
<td>HFP-square-root exception.</td>
</tr>
<tr>
<td>SS</td>
<td>SS instruction format.</td>
</tr>
<tr>
<td>SSE</td>
<td>SSE instruction format.</td>
</tr>
<tr>
<td>ST</td>
<td>PER storage-alteration event.</td>
</tr>
<tr>
<td>SU</td>
<td>PER store-using-real-address event.</td>
</tr>
</tbody>
</table>

- **SW**: Special-operation exception and space-switch event.
- **T**: Trace exceptions (which include trace table, addressing, and low-address protection).
- **U**: Condition code is unpredictable.
- **U1**: \( R_{1} \) field designates an access register unconditionally.
- **U2**: \( R_{2} \) field designates an access register unconditionally.
- **UB**: \( R_{1} \) and \( R_{2} \) fields designate access registers unconditionally, and \( B_{2} \) field designates an access register in the access-register mode.
- **WE**: Space-switch event.
- **Xi**: IEEE invalid-operation condition.
- **Xo**: IEEE overflow condition.
- **Xu**: IEEE underflow condition.
- **Xx**: IEEE inexact condition.
- **Xz**: IEEE division-by-zero condition.

**Z1**: Additional exceptions and events for PROGRAM CALL (which include ASX-translation, EX-translation, LX-translation, PC-translation-specification, special-operation, stack-full, and stack-specification exceptions and space-switch event).

**Z2**: Additional exceptions and events for PROGRAM TRANSFER (which include AFX-translation, ASX-translation, primary-authority, and special-operation exceptions and space-switch event).

**Z3**: Additional exceptions for SET SECONDARY ASN (which include AFX translation, ASX translation, secondary authority, and special operation).

**Z4**: Additional exceptions and events for PROGRAM RETURN (which include AFX-translation, ASX-translation, secondary-authority, special-operation, stack-empty, stack-operation, stack-specification, and stack-type exceptions and space-switch event).
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (extended BFP)</td>
<td>AXBR</td>
<td>RRE C</td>
<td>SP</td>
<td>B34A</td>
</tr>
<tr>
<td>ADD (long BFP)</td>
<td>ADBR</td>
<td>RRE C</td>
<td></td>
<td>B31A</td>
</tr>
<tr>
<td>ADD (long BFP)</td>
<td>ADBR</td>
<td>RXE C</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>ADD (short BFP)</td>
<td>AEBR</td>
<td>RRE C</td>
<td></td>
<td>B30A</td>
</tr>
<tr>
<td>ADD (short BFP)</td>
<td>AEBR</td>
<td>RXE C</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>AR</td>
<td>RR C</td>
<td></td>
<td>1A</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>AR</td>
<td>RX C</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>ADD (64&lt;32)</td>
<td>AGFR</td>
<td>RRE C</td>
<td>N</td>
<td>B918</td>
</tr>
<tr>
<td>ADD (64&lt;32)</td>
<td>AGFR</td>
<td>RXY C</td>
<td>LD</td>
<td>E318</td>
</tr>
<tr>
<td>ADD (64)</td>
<td>AGR</td>
<td>RRE C N</td>
<td></td>
<td>B908</td>
</tr>
<tr>
<td>ADD (64)</td>
<td>AG</td>
<td>RXY C N</td>
<td>A</td>
<td>E308</td>
</tr>
<tr>
<td>ADD DECIMAL</td>
<td>AP</td>
<td>SS C</td>
<td>A</td>
<td>1E</td>
</tr>
<tr>
<td>ADD HALFWORD</td>
<td>AH</td>
<td>RX C</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>ADD HALFWORD</td>
<td>AHY</td>
<td>RXY C</td>
<td>LD</td>
<td>BFA</td>
</tr>
<tr>
<td>ADD HALFWORD IMMEDIATE (32)</td>
<td>AHI</td>
<td>RI C</td>
<td>N</td>
<td>A7A</td>
</tr>
<tr>
<td>ADD HALFWORD IMMEDIATE (64)</td>
<td>AHI</td>
<td>RI C N</td>
<td></td>
<td>A7B</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>ALR</td>
<td>RR C</td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>AL</td>
<td>RX C</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>ADD LOGICAL (32)</td>
<td>ALY</td>
<td>RXY C</td>
<td>LD</td>
<td>B2</td>
</tr>
<tr>
<td>ADD LOGICAL (64)</td>
<td>ALGF</td>
<td>RRE C N</td>
<td></td>
<td>B91A</td>
</tr>
<tr>
<td>ADD LOGICAL (64)</td>
<td>ALGF</td>
<td>RXY C N</td>
<td>A</td>
<td>B31A</td>
</tr>
<tr>
<td>ADD LOGICAL (64)</td>
<td>ALGR</td>
<td>RRE C N</td>
<td></td>
<td>B90A</td>
</tr>
<tr>
<td>ADD LOGICAL (64)</td>
<td>ALGR</td>
<td>RXY C N</td>
<td>A</td>
<td>B30A</td>
</tr>
<tr>
<td>ADD LOGICAL WITH CARRY (32)</td>
<td>ALCR</td>
<td>RRE C N3</td>
<td></td>
<td>B998</td>
</tr>
<tr>
<td>ADD LOGICAL WITH CARRY (64)</td>
<td>ALCR</td>
<td>RXY C</td>
<td>N3</td>
<td>E398</td>
</tr>
<tr>
<td>ADD LOGICAL WITH CARRY (64)</td>
<td>ALCR</td>
<td>RXY C N</td>
<td></td>
<td>E388</td>
</tr>
<tr>
<td>ADD NORMALIZED (extended HFP)</td>
<td>AXR</td>
<td>RR C</td>
<td>SP</td>
<td>36</td>
</tr>
<tr>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>RR C</td>
<td></td>
<td>2A</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td>A</td>
<td>6A</td>
</tr>
<tr>
<td>ADD NORMALIZED (short HFP)</td>
<td>ADR</td>
<td>RX C</td>
<td></td>
<td>3A</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (long HFP)</td>
<td>AWR</td>
<td>RR C</td>
<td></td>
<td>2E</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (long HFP)</td>
<td>AWR</td>
<td>RX C</td>
<td>A</td>
<td>6E</td>
</tr>
<tr>
<td>ADD UNNORMALIZED (short HFP)</td>
<td>AUR</td>
<td>RR C</td>
<td></td>
<td>3E</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>NR</td>
<td>RR C</td>
<td>A</td>
<td>14</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>N</td>
<td>RX C</td>
<td>A</td>
<td>14</td>
</tr>
<tr>
<td>ADD (64)</td>
<td>N</td>
<td>RXY C</td>
<td>LD</td>
<td>E354</td>
</tr>
<tr>
<td>ADD (64)</td>
<td>N</td>
<td>RRE C</td>
<td>N</td>
<td>B980</td>
</tr>
</tbody>
</table>

Figure  B-1 (Part 1 of 15). Instructions Arranged by Name

Appendix B. Lists of Instructions  B-3
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND IMMEDIATE (high high)</td>
<td>NIHH</td>
<td>RI C N</td>
<td>A54</td>
<td>7-21</td>
</tr>
<tr>
<td>AND IMMEDIATE (high low)</td>
<td>NIHL</td>
<td>RI C N</td>
<td>A55</td>
<td>7-21</td>
</tr>
<tr>
<td>AND IMMEDIATE (low high)</td>
<td>NILH</td>
<td>RI C N</td>
<td>A56</td>
<td>7-21</td>
</tr>
<tr>
<td>AND IMMEDIATE (low low)</td>
<td>NILL</td>
<td>RI C N</td>
<td>A57</td>
<td>7-21</td>
</tr>
<tr>
<td>BRANCH AND LINK</td>
<td>BAL</td>
<td>RR</td>
<td>45</td>
<td>7-22</td>
</tr>
<tr>
<td>BRANCH AND SAVE</td>
<td>BASR</td>
<td>RX</td>
<td>46</td>
<td>7-23</td>
</tr>
<tr>
<td>BRANCH AND SAVE AND SET MODE</td>
<td>BASM</td>
<td>RR</td>
<td>40</td>
<td>7-23</td>
</tr>
<tr>
<td>BRANCH AND SET AUTHORITY</td>
<td>BSA</td>
<td>RR</td>
<td>40</td>
<td>7-23</td>
</tr>
<tr>
<td>BRANCH AND SET MODE</td>
<td>BSM</td>
<td>RR</td>
<td>80</td>
<td>7-24</td>
</tr>
<tr>
<td>BRANCH AND STACK</td>
<td>BAKR</td>
<td>RR</td>
<td>87</td>
<td>7-25</td>
</tr>
<tr>
<td>BRANCH IN SUBSPACE GROUP</td>
<td>BSJ</td>
<td>RR</td>
<td>47</td>
<td>7-25</td>
</tr>
<tr>
<td>BRANCH ON CONDITION</td>
<td>BCR</td>
<td>RR</td>
<td>07</td>
<td>7-25</td>
</tr>
<tr>
<td>BRANCH ON COUNT (32)</td>
<td>BCTR</td>
<td>RR</td>
<td>06</td>
<td>7-25</td>
</tr>
<tr>
<td>BRANCH ON COUNT (64)</td>
<td>BCTG</td>
<td>RRE N</td>
<td>46</td>
<td>7-26</td>
</tr>
<tr>
<td>BRANCH ON INDEX (32)</td>
<td>BXH</td>
<td>RS</td>
<td>86</td>
<td>7-27</td>
</tr>
<tr>
<td>BRANCH ON INDEX (64)</td>
<td>BXHG</td>
<td>RRE N</td>
<td>87</td>
<td>7-27</td>
</tr>
<tr>
<td>BRANCH ON INDEX LOW OR EQUAL (32)</td>
<td>BXLE</td>
<td>RS</td>
<td>86</td>
<td>7-27</td>
</tr>
<tr>
<td>BRANCH ON INDEX LOW OR EQUAL (64)</td>
<td>BXLEG</td>
<td>RS</td>
<td>87</td>
<td>7-27</td>
</tr>
<tr>
<td>BRANCH RELATIVE AND SAVE</td>
<td>BRAS</td>
<td>RI</td>
<td>B</td>
<td>7-28</td>
</tr>
<tr>
<td>BRANCH RELATIVE AND SAVE LONG</td>
<td>BRASL</td>
<td>RIL N3</td>
<td>B</td>
<td>7-28</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON CONDITION</td>
<td>BRC</td>
<td>RI</td>
<td>B</td>
<td>7-29</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON CONDITION LONG</td>
<td>BRCL</td>
<td>RIL N3</td>
<td>B</td>
<td>7-29</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON COUNT (32)</td>
<td>BRCT</td>
<td>RI</td>
<td>B</td>
<td>7-29</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON COUNT (64)</td>
<td>BRCTG</td>
<td>RI</td>
<td>B</td>
<td>7-29</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON INDEX (32)</td>
<td>BRXH</td>
<td>RSI</td>
<td>B</td>
<td>7-30</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON INDEX (64)</td>
<td>BRXHG</td>
<td>RIE N</td>
<td>B</td>
<td>7-30</td>
</tr>
<tr>
<td>BRANCH RELATIVE ON INDEX L OR E (32)</td>
<td>BRXLG</td>
<td>RIE N</td>
<td>B</td>
<td>7-30</td>
</tr>
<tr>
<td>CANCEL SUBCHANNEL</td>
<td>XSCH</td>
<td>S C</td>
<td>B</td>
<td>14-4</td>
</tr>
<tr>
<td>CHECKSUM</td>
<td>CKSM</td>
<td>RRE C</td>
<td>R2</td>
<td>31-7</td>
</tr>
<tr>
<td>CIPHER MESSAGE</td>
<td>KM</td>
<td>RRE C MS</td>
<td>A66</td>
<td>7-35</td>
</tr>
<tr>
<td>CIPHER MESSAGE WITH CHAINING</td>
<td>KMC</td>
<td>RRE C MS</td>
<td>A66</td>
<td>7-35</td>
</tr>
<tr>
<td>CLEAR SUBCHANNEL</td>
<td>CSCH</td>
<td>S C</td>
<td>A66</td>
<td>7-35</td>
</tr>
<tr>
<td>COMPARE (extended BFP)</td>
<td>CXBR</td>
<td>RRE C</td>
<td>B</td>
<td>19-23</td>
</tr>
<tr>
<td>COMPARE (extended HFP)</td>
<td>CXR</td>
<td>RRE C</td>
<td>B</td>
<td>18-10</td>
</tr>
<tr>
<td>COMPARE (long BFP)</td>
<td>CDBR</td>
<td>RRE C</td>
<td>B</td>
<td>19-23</td>
</tr>
<tr>
<td>COMPARE (long BFP)</td>
<td>CDBR</td>
<td>RRE C</td>
<td>B</td>
<td>19-23</td>
</tr>
<tr>
<td>COMPARE (long HFP)</td>
<td>CDBR</td>
<td>RRE C</td>
<td>B</td>
<td>19-23</td>
</tr>
<tr>
<td>COMPARE (long HFP)</td>
<td>CDBR</td>
<td>RRE C</td>
<td>B</td>
<td>19-23</td>
</tr>
<tr>
<td>COMPARE (short BFP)</td>
<td>CEBR</td>
<td>RRE C</td>
<td>B</td>
<td>18-10</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 2 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARE (short BFP)</td>
<td>CEB</td>
<td>RXE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE (short HFP)</td>
<td>CER</td>
<td>RR C</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>COMPARE (short HFP)</td>
<td>CE</td>
<td>RX C</td>
<td>A</td>
<td>Da</td>
</tr>
<tr>
<td>COMPARE (32)</td>
<td>CR</td>
<td>RX C</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>COMPARE (32)</td>
<td>C</td>
<td>RX C</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>COMPARE (64&lt;32)</td>
<td>CY</td>
<td>RX C LD</td>
<td>A</td>
<td>II</td>
</tr>
<tr>
<td>COMPARE AND FORM CODEWORD</td>
<td>CFC</td>
<td>S C</td>
<td>A SP</td>
<td>GM</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (extended BFP)</td>
<td>KXBR</td>
<td>RRE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (long BFP)</td>
<td>KDBR</td>
<td>RRE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SIGNAL (short BFP)</td>
<td>KEB</td>
<td>RXE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>COMPARE AND SWAP (32)</td>
<td>CS</td>
<td>RS C</td>
<td>A SP</td>
<td>$</td>
</tr>
<tr>
<td>COMPARE AND SWAP (32)</td>
<td>CSY</td>
<td>RSY C LD</td>
<td>A SP</td>
<td>$</td>
</tr>
<tr>
<td>COMPARE AND SWAP AND PURGE</td>
<td>CSP</td>
<td>RRE C N</td>
<td>A SP</td>
<td>$</td>
</tr>
<tr>
<td>COMPARE AND SWAP AND PURGE</td>
<td>CSPG</td>
<td>RRE C DE</td>
<td>P A¹ SP</td>
<td>$</td>
</tr>
<tr>
<td>COMPARE DECIMAL</td>
<td>CP</td>
<td>SS C</td>
<td>A</td>
<td>Dd</td>
</tr>
<tr>
<td>COMPARE HALFWORD</td>
<td>CH</td>
<td>RX C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE HALFWORD</td>
<td>CHY</td>
<td>RXY C LD</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE HALFWORD IMMEDIATE (32)</td>
<td>CHI</td>
<td>RI C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE HALFWORD IMMEDIATE (64)</td>
<td>CGHI</td>
<td>RI C N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (character)</td>
<td>CLC</td>
<td>SS C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (immediate)</td>
<td>CLI</td>
<td>SI C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (immediate)</td>
<td>CLIY</td>
<td>SIY C LD</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (32)</td>
<td>CLR</td>
<td>RR C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (32)</td>
<td>CL</td>
<td>RX C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (32)</td>
<td>CLY</td>
<td>RX Y C LD</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>CLGFR</td>
<td>RRE C N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>CLGF</td>
<td>RXY C N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (64)</td>
<td>CLGR</td>
<td>RRE C N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL (64)</td>
<td>CLG</td>
<td>RXY C N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL C. UNDER MASK (high)</td>
<td>CLMH</td>
<td>RXS C N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL C. UNDER MASK (low)</td>
<td>CLM</td>
<td>RS C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL C. UNDER MASK (low)</td>
<td>CLMY</td>
<td>RSY C LD</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL LONG</td>
<td>CLCL</td>
<td>RR C</td>
<td>A SP</td>
<td>II</td>
</tr>
<tr>
<td>COMPARE LOGICAL LONG EXTENDED</td>
<td>CLCLE</td>
<td>RS C</td>
<td>A SP</td>
<td></td>
</tr>
<tr>
<td>COMPARE LOGICAL LONG UNICODE</td>
<td>CLCLU</td>
<td>RSY C E2</td>
<td>A SP</td>
<td></td>
</tr>
</tbody>
</table>

Figure B-1 (Part 3 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARE LOGICAL STRING</td>
<td>CLST</td>
<td>RRE C A SP</td>
<td>G0</td>
<td>R1, R2</td>
</tr>
<tr>
<td>COMPARE UNTIL SUBSTRING EQUAL</td>
<td>CUSE</td>
<td>RRE C A SP</td>
<td>II Dd GM</td>
<td>R1, R2</td>
</tr>
<tr>
<td>COMPRESSION CALL</td>
<td>CMPSC</td>
<td>RRE C A SP</td>
<td>II GM</td>
<td>ST</td>
</tr>
<tr>
<td>COMPUTE INTERMEDIATE MESSAGE DIGEST</td>
<td>KIMD</td>
<td>RRE C MS A SP</td>
<td>GM I1</td>
<td>ST</td>
</tr>
<tr>
<td>COMPUTE LAST MESSAGE DIGEST</td>
<td>KLMD</td>
<td>RRE C MS A SP</td>
<td>GM I1</td>
<td>ST</td>
</tr>
<tr>
<td>COMPUTE MESSAGE AUTHENTICATION CODE</td>
<td>KMAC</td>
<td>RRE C MS A SP</td>
<td>GM I1</td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT BFP TO HFP (long)</td>
<td>THDR</td>
<td>RRE C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT BFP TO HFP (short to long)</td>
<td>THDER</td>
<td>RRE C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to ext. BFP)</td>
<td>CXFBR</td>
<td>RRE SP</td>
<td>Db</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to ext. HFP)</td>
<td>CXFR</td>
<td>RRE SP</td>
<td>Db</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to long BFP)</td>
<td>CDFBR</td>
<td>RRE SP</td>
<td>Db</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to long HFP)</td>
<td>CDFR</td>
<td>RRE SP</td>
<td>Db</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to short BFP)</td>
<td>CEFBR</td>
<td>RRE SP</td>
<td>Db</td>
<td>Xx</td>
</tr>
<tr>
<td>CONVERT FROM FIXED (32 to short HFP)</td>
<td>CEFR</td>
<td>RRE SP</td>
<td>Db</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to ext. BFP)</td>
<td>CXGBR</td>
<td>RRE N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to ext. HFP)</td>
<td>CXGR</td>
<td>RRE N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to long BFP)</td>
<td>CGXBR</td>
<td>RRE N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to long HFP)</td>
<td>CGXR</td>
<td>RRE N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to short BFP)</td>
<td>CEBR</td>
<td>RRE N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT FROM FIXED (64 to short HFP)</td>
<td>CER</td>
<td>RRE N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT HFP TO BFP (long to short)</td>
<td>TBEDR</td>
<td>RRF C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT HFP TO BFP (long)</td>
<td>TBDR</td>
<td>RRF C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO BINARY (32)</td>
<td>CVB</td>
<td>RX A</td>
<td>Dd IK</td>
<td>B2</td>
</tr>
<tr>
<td>CONVERT TO BINARY (32)</td>
<td>CVBY</td>
<td>RXY LD A</td>
<td>Dd IK</td>
<td>B2</td>
</tr>
<tr>
<td>CONVERT TO BINARY (64)</td>
<td>CVBG</td>
<td>RXY N A</td>
<td>Dd IK</td>
<td>B2</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (32)</td>
<td>CVD</td>
<td>RX A</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (32)</td>
<td>CVDY</td>
<td>RXY LD A</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT TO DECIMAL (64)</td>
<td>CVDG</td>
<td>RXY N A</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT TO FIXED (ext. BFP to 32)</td>
<td>CFXBR</td>
<td>RRF C SP</td>
<td>Db Xi Xx</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (ext. BFP to 64)</td>
<td>CGXBR</td>
<td>RRF C N SP</td>
<td>Db Xi Xx</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (ext. HFP to 32)</td>
<td>CFXR</td>
<td>RRF C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (ext. HFP to 64)</td>
<td>CGXBR</td>
<td>RRF C N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (long BFP to 32)</td>
<td>CFBFR</td>
<td>RRF C SP</td>
<td>Db Xi Xx</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (long BFP to 64)</td>
<td>CGFBR</td>
<td>RRF C N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (long HFP to 32)</td>
<td>CGD</td>
<td>RRF C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (long HFP to 64)</td>
<td>CGER</td>
<td>RRF C N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (short BFP to 32)</td>
<td>CFER</td>
<td>RRF C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (short BFP to 64)</td>
<td>CGER</td>
<td>RRF C N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (short HFP to 32)</td>
<td>CFER</td>
<td>RRF C SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT TO FIXED (short HFP to 64)</td>
<td>CGER</td>
<td>RRF C N SP</td>
<td>Da</td>
<td></td>
</tr>
<tr>
<td>CONVERT UNCODE TO UTF-8</td>
<td>CUUTF</td>
<td>RRE C A SP</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-32</td>
<td>CU24</td>
<td>RRE C E3 A SP</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-8</td>
<td>CU21</td>
<td>RRE C E3 A SP</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT UTF-32 TO UTF-16</td>
<td>CU42</td>
<td>RRE C E3 A SP</td>
<td></td>
<td>ST</td>
</tr>
<tr>
<td>CONVERT UTF-32 TO UTF-8</td>
<td>CU41</td>
<td>RRE C E3 A SP</td>
<td></td>
<td>ST</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 4 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVERT UTF-8 TO UNICODE</td>
<td>CUTFU</td>
<td>RRE C</td>
<td>ST</td>
<td>R2 R3</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UTF-16</td>
<td>CUTFU</td>
<td>RRE C E3</td>
<td>ST</td>
<td>R2 R3</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UTF-32</td>
<td>CUTFU</td>
<td>RRE C E3</td>
<td>ST</td>
<td>R2 R3</td>
</tr>
<tr>
<td>COPY ACCESS</td>
<td>CPYA</td>
<td>RRE DM P DM</td>
<td>ST</td>
<td>U1 U2</td>
</tr>
<tr>
<td>DIAGNOSE</td>
<td></td>
<td></td>
<td>MD</td>
<td>7-10</td>
</tr>
<tr>
<td>DIVIDE (extended BFP)</td>
<td>DXBR</td>
<td>RRE SP</td>
<td>B34D</td>
<td>19-29</td>
</tr>
<tr>
<td>DIVIDE (extended HFP)</td>
<td>DXR</td>
<td>RRE SP</td>
<td>B220</td>
<td>18-12</td>
</tr>
<tr>
<td>DIVIDE (long BFP)</td>
<td>DDBR</td>
<td>RRE Sp</td>
<td>B31D</td>
<td>19-29</td>
</tr>
<tr>
<td>DIVIDE (long BFP)</td>
<td>DDBR</td>
<td>RRE A</td>
<td>E0D</td>
<td>19-29</td>
</tr>
<tr>
<td>DIVIDE (long HFP)</td>
<td>DDR</td>
<td>RRE Da E0 Fk</td>
<td>2D</td>
<td>18-12</td>
</tr>
<tr>
<td>DIVIDE (long HFP)</td>
<td>DDR</td>
<td>RX Da E0 Fk</td>
<td>B2</td>
<td>18-12</td>
</tr>
<tr>
<td>DIVIDE (short BFP)</td>
<td>DEBR</td>
<td>RRE Da E0 Fk</td>
<td>B30D</td>
<td>19-29</td>
</tr>
<tr>
<td>DIVIDE (short BFP)</td>
<td>DEB</td>
<td>RX Da E0 Fk</td>
<td>B0D</td>
<td>19-29</td>
</tr>
<tr>
<td>DIVIDE (short HFP)</td>
<td>DER</td>
<td>RX Da E0 Fk</td>
<td>3D</td>
<td>18-12</td>
</tr>
<tr>
<td>DIVIDE (short HFP)</td>
<td>DE</td>
<td>RX Da E0 Fk</td>
<td>7D</td>
<td>18-12</td>
</tr>
<tr>
<td>DIVIDE (32&lt;64)</td>
<td>DR</td>
<td>RRE Sp</td>
<td>1D</td>
<td>7-116</td>
</tr>
<tr>
<td>DIVIDE (32&lt;64)</td>
<td>D</td>
<td>RX Da E0 Fk</td>
<td>B2</td>
<td>7-116</td>
</tr>
<tr>
<td>DIVIDE DECIMAL</td>
<td></td>
<td></td>
<td></td>
<td>7-116</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (32&lt;64)</td>
<td>DLR</td>
<td>RRE N3 Sp</td>
<td>B997</td>
<td>7-117</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (32&lt;64)</td>
<td>DL</td>
<td>RX N3 A Sp</td>
<td>E397</td>
<td>7-117</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>DLR</td>
<td>RRE N Sp</td>
<td>B987</td>
<td>7-117</td>
</tr>
<tr>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>DLG</td>
<td>RRE N A Sp</td>
<td>E387</td>
<td>7-117</td>
</tr>
<tr>
<td>DIVIDE SINGLE (64)</td>
<td>DSGF</td>
<td>RRE N A Sp</td>
<td>E13D</td>
<td>7-118</td>
</tr>
<tr>
<td>DIVIDE SINGLE (64)</td>
<td>DSGR</td>
<td>RRE N A Sp</td>
<td>B90D</td>
<td>7-117</td>
</tr>
<tr>
<td>DIVIDE TO INTEGER (long BFP)</td>
<td>DIDBR</td>
<td>RRF C Sp</td>
<td>B30D</td>
<td>19-29</td>
</tr>
<tr>
<td>DIVIDE TO INTEGER (short BFP)</td>
<td>DIEBR</td>
<td>RRF C Sp</td>
<td>B353</td>
<td>19-29</td>
</tr>
<tr>
<td>EDIT</td>
<td>ED</td>
<td>RXY A Dd G1</td>
<td>ST</td>
<td>B1 B2</td>
</tr>
<tr>
<td>EDIT AND MARK</td>
<td>EDMK</td>
<td>RXY A Dd G1</td>
<td>ST</td>
<td>B1 B2</td>
</tr>
<tr>
<td>EXCLUSIVE OR (character)</td>
<td>XC</td>
<td>SS C A</td>
<td>ST</td>
<td>B1 B2</td>
</tr>
<tr>
<td>EXCLUSIVE OR (immediate)</td>
<td>XI</td>
<td>SI C A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>EXCLUSIVE OR (immediate)</td>
<td>XY</td>
<td>SY C LD</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>EXCLUSIVE OR (32)</td>
<td>XR</td>
<td>RR C A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>EXCLUSIVE OR (32)</td>
<td>X</td>
<td>RX C A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>EXCLUSIVE OR (32)</td>
<td>XY</td>
<td>RXY C LD A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>EXCLUSIVE OR (64)</td>
<td>XG</td>
<td>RRE C N A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>EXCLUSIVE OR (64)</td>
<td>X</td>
<td>RX N A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>EX</td>
<td>RX A Sp EX</td>
<td>U2</td>
<td>B24F</td>
</tr>
<tr>
<td>EXTRACT ACCESS</td>
<td>EAR</td>
<td>RRE Db</td>
<td>B99D</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT FPC</td>
<td>EFPC</td>
<td>RRE Db Q</td>
<td>B38C</td>
<td>19-33</td>
</tr>
<tr>
<td>EXTRACT PRIMARY ASN</td>
<td>EPAR</td>
<td>RRE SO</td>
<td>B226</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT PRIMARY ASN AND INSTANCE</td>
<td>EPAIR</td>
<td>RRE RA QA</td>
<td>B99A</td>
<td>10-21</td>
</tr>
<tr>
<td>EXTRACT PSW</td>
<td>EPSW</td>
<td>RRE N3</td>
<td>B98D</td>
<td>7-120</td>
</tr>
</tbody>
</table>

*Figure B-1 (Part 5 of 15). Instructions Arranged by Name*
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTRACT SECONDARY ASN</td>
<td>ESAR</td>
<td>RRE RA Q</td>
<td>B227</td>
<td>10-22</td>
</tr>
<tr>
<td>EXTRACT SECONDARY ASN AND INSTANCE</td>
<td>ESAIR</td>
<td>RRE RA Q</td>
<td>U1 U2</td>
<td>10-22</td>
</tr>
<tr>
<td>EXTRACT STACKED REGISTERS (32)</td>
<td>EREG</td>
<td>RRE N A(^1) SE</td>
<td>B249</td>
<td>10-22</td>
</tr>
<tr>
<td>EXTRACT STACKED REGISTERS (64)</td>
<td>EREGG</td>
<td>RRE C A(^1) SP</td>
<td>B90E</td>
<td>10-22</td>
</tr>
<tr>
<td>EXTRACT STACKED STATE</td>
<td>ESTA</td>
<td></td>
<td>B24A</td>
<td>10-24</td>
</tr>
<tr>
<td>HALT SUBCHANNEL</td>
<td>HSCH</td>
<td>S C P</td>
<td>B231</td>
<td>14-6</td>
</tr>
<tr>
<td>HALVE (long HFP)</td>
<td>HDR</td>
<td>RRE</td>
<td>24</td>
<td>18-13</td>
</tr>
<tr>
<td>HALVE (short HFP)</td>
<td>HER</td>
<td>RRE</td>
<td>34</td>
<td>18-13</td>
</tr>
<tr>
<td>INSERT ADDRESS SPACE CONTROL</td>
<td>IAC</td>
<td>RRE</td>
<td>B224</td>
<td>10-27</td>
</tr>
<tr>
<td>INSERT CHARACTER</td>
<td>IC</td>
<td>RX A</td>
<td>B4</td>
<td>7-121</td>
</tr>
<tr>
<td>INSERT CHARACTERS UNDER MASK (high)</td>
<td>ICMH</td>
<td>RSY C N A(^1) G2</td>
<td>B2</td>
<td>7-121</td>
</tr>
<tr>
<td>INSERT CHARACTERS UNDER MASK (low)</td>
<td>ICM</td>
<td>RS C A(^1) G2</td>
<td>B2</td>
<td>7-121</td>
</tr>
<tr>
<td>INSERT CHARACTERS UNDER MASK (low)</td>
<td>ICMY</td>
<td>RSY C LD A(^1) G2</td>
<td>B2</td>
<td>7-121</td>
</tr>
<tr>
<td>INSERT IMMEDIATE (high high)</td>
<td>IIHL</td>
<td>RI N</td>
<td>B222</td>
<td>7-122</td>
</tr>
<tr>
<td>INSERT IMMEDIATE (low high)</td>
<td>IIIL</td>
<td>RI N</td>
<td>B222</td>
<td>7-122</td>
</tr>
<tr>
<td>INSERT IMMEDIATE (low low)</td>
<td>IIIL</td>
<td>RI N</td>
<td>B222</td>
<td>7-122</td>
</tr>
<tr>
<td>INSERT PROGRAM MASK</td>
<td>IPM</td>
<td>RRE</td>
<td>B2B</td>
<td>7-122</td>
</tr>
<tr>
<td>INSERT PSW KEY</td>
<td>IPK</td>
<td>S Q</td>
<td>A51</td>
<td>7-122</td>
</tr>
<tr>
<td>INSERT STORAGE KEY EXTENDED</td>
<td>ISKE</td>
<td>RRE P A(^1) Q A(^1) SO</td>
<td>B299</td>
<td>10-28</td>
</tr>
<tr>
<td>INSERT STORAGE KEY EXTENDED</td>
<td>IVSK</td>
<td>RRE P A(^1) Q A(^1) Da $</td>
<td>B223</td>
<td>10-29</td>
</tr>
<tr>
<td>INVALIDATE DAT TABLE ENTRY</td>
<td>IDTE</td>
<td>RRF DE P A(^1) Da $</td>
<td>B9BE</td>
<td>10-30</td>
</tr>
<tr>
<td>INVALIDATE PAGE TABLE ENTRY</td>
<td>IPTE</td>
<td>RRE P A(^1) Da $</td>
<td>B221</td>
<td>10-34</td>
</tr>
<tr>
<td>LOAD (extended)</td>
<td>LXR</td>
<td>RR A</td>
<td>B365</td>
<td>9-10</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>LDR</td>
<td>RR A</td>
<td>28</td>
<td>9-10</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>LD</td>
<td>RX A</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LOAD (long)</td>
<td>LDY</td>
<td>RXY LD A</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LOAD (short)</td>
<td>LER</td>
<td>RR A</td>
<td>38</td>
<td>9-10</td>
</tr>
<tr>
<td>LOAD (short)</td>
<td>LE</td>
<td>RX A</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LOAD (short)</td>
<td>LEY</td>
<td>RXY LD A</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LOAD (32)</td>
<td>LR</td>
<td>RR A</td>
<td>B2</td>
<td>9-11</td>
</tr>
<tr>
<td>LOAD (32)</td>
<td>L</td>
<td>RX A</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD (32)</td>
<td>LY</td>
<td>RXY LD A</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD (64&lt;32)</td>
<td>LGFR</td>
<td>RRE N</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD (64)</td>
<td>LGF</td>
<td>RXY N A</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD (64)</td>
<td>LGR</td>
<td>RRE N A</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD (64)</td>
<td>LG</td>
<td>RXY N A</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD ACCESS MULTIPLE</td>
<td>LAM</td>
<td>RS A SP</td>
<td>9A</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD ACCESS MULTIPLE</td>
<td>LMY</td>
<td>RSY LD A SP</td>
<td>EB9A</td>
<td>7-123</td>
</tr>
<tr>
<td>LOAD ADDRESS</td>
<td>LA</td>
<td>RX</td>
<td>41</td>
<td>7-124</td>
</tr>
<tr>
<td>LOAD ADDRESS</td>
<td>LAY</td>
<td>RXY LD</td>
<td>E371</td>
<td>7-124</td>
</tr>
<tr>
<td>LOAD ADDRESS EXTENDED</td>
<td>LAE</td>
<td>RX</td>
<td>U1 BP</td>
<td>7-124</td>
</tr>
<tr>
<td>LOAD ADDRESS RELATIVE LONG</td>
<td>LRL</td>
<td>RIL N3</td>
<td>C00</td>
<td>7-125</td>
</tr>
<tr>
<td>LOAD ADDRESS SPACE PARAMETERS</td>
<td>LASP</td>
<td>SSE C P A(^1) SP SO</td>
<td>E500</td>
<td>10-36</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 6 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD AND TEST (extended BFP)</td>
<td>LTXBR</td>
<td>RRE C</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD AND TEST (extended HFP)</td>
<td>LTBR</td>
<td>RRE C</td>
<td>SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD AND TEST (long BFP)</td>
<td>LTOBR</td>
<td>RRE C</td>
<td></td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD AND TEST (long HFP)</td>
<td>LDR</td>
<td>RRE C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD AND TEST (short BFP)</td>
<td>LTEBR</td>
<td>RRE C</td>
<td></td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD AND TEST (short HFP)</td>
<td>LTER</td>
<td>RR C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD AND TEST (32)</td>
<td>LTR</td>
<td>RR C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD AND TEST (64&lt;32)</td>
<td>LTGFR</td>
<td>RRE C N</td>
<td></td>
<td>IF</td>
</tr>
<tr>
<td>LOAD AND TEST (64)</td>
<td>LTR</td>
<td>RR C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD BYTE (32)</td>
<td>LGB</td>
<td>RX Y LD</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD BYTE (64)</td>
<td>LGB</td>
<td>RX Y LD</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (extended BFP)</td>
<td>LCXB</td>
<td>RR C</td>
<td>SP</td>
<td>Db</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (extended HFP)</td>
<td>LCXBR</td>
<td>RRE C N</td>
<td>SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (long BFP)</td>
<td>LCBR</td>
<td>RRE C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (long HFP)</td>
<td>LCR</td>
<td>RR C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (short BFP)</td>
<td>LCEBR</td>
<td>RRE C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (short HFP)</td>
<td>LCEBR</td>
<td>RRE C</td>
<td></td>
<td>Da</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (32)</td>
<td>LCR</td>
<td>RR C</td>
<td></td>
<td>IF</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (64&lt;32)</td>
<td>LCGFR</td>
<td>RRE C N</td>
<td>SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (64)</td>
<td>LCGR</td>
<td>RRE C N</td>
<td>SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD CONTROL (32)</td>
<td>LCTL</td>
<td>RS</td>
<td>P A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD CONTROL (64)</td>
<td>LCTL</td>
<td>RS</td>
<td>P A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD FP INTEGER (extended BFP)</td>
<td>FIXBR</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD FP INTEGER (extended HFP)</td>
<td>FIXR</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD FP INTEGER (long BFP)</td>
<td>FIODR</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD FP INTEGER (long HFP)</td>
<td>FIODR</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD FP INTEGER (short BFP)</td>
<td>FIDR</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD FP INTEGER (short HFP)</td>
<td>FIDR</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD FPC</td>
<td>LFPC</td>
<td>S</td>
<td>A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD HALFWORD (32)</td>
<td>LH</td>
<td>RX</td>
<td>A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD HALFWORD (64)</td>
<td>LH</td>
<td>RX</td>
<td>A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD HALFWORD IMMEDIATE (32)</td>
<td>LHI</td>
<td>RI</td>
<td>A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD HALFWORD IMMEDIATE (64)</td>
<td>LGHI</td>
<td>RI</td>
<td>A</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD LENGTHENED (long to ext. BFP)</td>
<td>LDB</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (long to ext. HFP)</td>
<td>LDB</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (long to ext. HFP)</td>
<td>LDB</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. BFP)</td>
<td>LEBR</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. BFP)</td>
<td>LEBR</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. BFP)</td>
<td>LEBR</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>LXR</td>
<td>RXE</td>
<td>A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>LXR</td>
<td>RXE</td>
<td>A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>LXR</td>
<td>RXE</td>
<td>A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>LDEBR</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>LDEB</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>LDEB</td>
<td>RXE</td>
<td>A SP</td>
<td>Db Xi</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>LDER</td>
<td>RXE</td>
<td>A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>LDER</td>
<td>RXE</td>
<td>A SP</td>
<td>Da</td>
</tr>
<tr>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>LDER</td>
<td>RXE</td>
<td>A SP</td>
<td>Da</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 7 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>LDE RXE A Da</td>
<td>B2 ED24</td>
<td>18-15</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL (64&lt;32)</td>
<td>LLGFR RRE N A</td>
<td>B2 E316</td>
<td>7-128</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL CHARACTER</td>
<td>LLGC RXY N A</td>
<td>B2 E390</td>
<td>7-128</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL HALFWORD</td>
<td>LLGH RXY N A</td>
<td>B2 E391</td>
<td>7-128</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE (high high)</td>
<td>LLIHH RI N</td>
<td>A5C</td>
<td>7-128</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE (high low)</td>
<td>LLIHL RI N</td>
<td>A5D</td>
<td>7-128</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE (low high)</td>
<td>LLILH RI N</td>
<td>A5E</td>
<td>7-128</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL IMMEDIATE (low low)</td>
<td>LLILL RI N</td>
<td>A5F</td>
<td>7-128</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>LLGTR RRE N</td>
<td>B917</td>
<td>7-129</td>
<td></td>
</tr>
<tr>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>LLGT RXY N A</td>
<td>B2 EB96</td>
<td>7-129</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE (32)</td>
<td>LM RX C Da</td>
<td>B2 EB98</td>
<td>7-129</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE (64)</td>
<td>LMG RXY LD A</td>
<td>B2 EB04</td>
<td>7-129</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE DISJOINT</td>
<td>LMD SS N A</td>
<td>B2 EF</td>
<td>7-130</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE HIGH</td>
<td>LMH RXY N A</td>
<td>B2 98</td>
<td>7-129</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE IMMEDIATE (long BFP)</td>
<td>LNXBR RRE C SP Da</td>
<td>B2 B341</td>
<td>19-37</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE IMMEDIATE (long HFP)</td>
<td>LNXR RRE C SP Da</td>
<td>B2 B361</td>
<td>18-16</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE IMMEDIATE (long BFP)</td>
<td>LNDR RR C Da</td>
<td>B2 B311</td>
<td>19-37</td>
<td></td>
</tr>
<tr>
<td>LOAD MULTIPLE IMMEDIATE (long HFP)</td>
<td>LNDGR RRE C N Da</td>
<td>B2 B310</td>
<td>18-16</td>
<td></td>
</tr>
<tr>
<td>LOAD PAIR FROM QUADWORD</td>
<td>LPQ RXY N A SP Da</td>
<td>B2 B300</td>
<td>19-38</td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE (extended BFP)</td>
<td>LPXBR RRE C SP Da</td>
<td>B2 B340</td>
<td>19-38</td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE (extended HFP)</td>
<td>LPXR RRE C SP Da</td>
<td>B2 B360</td>
<td>18-16</td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE (long BFP)</td>
<td>LPDBR RRE C Da</td>
<td>B2 B310</td>
<td>19-38</td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE (long HFP)</td>
<td>LPDR RR C Da</td>
<td>B2 B300</td>
<td>18-16</td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE (short BFP)</td>
<td>LPEBR RRE C Da</td>
<td>B2 B300</td>
<td>19-38</td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE (short HFP)</td>
<td>LPER RR C Da</td>
<td>B2 B910</td>
<td>18-16</td>
<td></td>
</tr>
<tr>
<td>LOAD POSITIVE (short BFP)</td>
<td>LPFGFR RRE C N Da</td>
<td>B2 B910</td>
<td>17-31</td>
<td></td>
</tr>
<tr>
<td>LOAD PSW</td>
<td>LPSW S L PA SP</td>
<td>B2 82</td>
<td>10-45</td>
<td></td>
</tr>
<tr>
<td>LOAD PSW EXTENDED</td>
<td>LPSWE S L PA SP</td>
<td>B2 B282</td>
<td>10-46</td>
<td></td>
</tr>
<tr>
<td>LOAD REAL ADDRESS (32)</td>
<td>LRA RX C PA1 SO</td>
<td>B2 BP B1</td>
<td>10-47</td>
<td></td>
</tr>
<tr>
<td>LOAD REAL ADDRESS (32)</td>
<td>LRAY RXY C LD PA1 SO</td>
<td>B2 BP B313</td>
<td>10-47</td>
<td></td>
</tr>
<tr>
<td>LOAD REAL ADDRESS (64)</td>
<td>LRACT RXY C N PA1 SO</td>
<td>B2 BP B303</td>
<td>10-47</td>
<td></td>
</tr>
<tr>
<td>LOAD REVERSED (16)</td>
<td>LRVH RXY N3 A</td>
<td>B2 E31F</td>
<td>7-132</td>
<td></td>
</tr>
<tr>
<td>LOAD REVERSED (32)</td>
<td>LRVN RXY N3 A</td>
<td>B2 891F</td>
<td>7-132</td>
<td></td>
</tr>
<tr>
<td>LOAD REVERSED (32)</td>
<td>LRY RXY N3 A</td>
<td>B2 E31E</td>
<td>7-132</td>
<td></td>
</tr>
<tr>
<td>LOAD REVERSED (64)</td>
<td>LRVGR RRE N A</td>
<td>B2 890F</td>
<td>7-132</td>
<td></td>
</tr>
<tr>
<td>LOAD REVERSED (64)</td>
<td>LRVG RXY N A</td>
<td>B2 E30F</td>
<td>7-132</td>
<td></td>
</tr>
</tbody>
</table>

Figure B-1 (Part 8 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-monic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD ROUNDED (extended to long BFP)</td>
<td>LDXBR</td>
<td>RRE</td>
<td>Db Xi</td>
<td>B345</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to long BFP)</td>
<td>LDXR</td>
<td>RR</td>
<td>Da EO</td>
<td>25</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to long BFP)</td>
<td>LDRD</td>
<td>RR</td>
<td>Da EO</td>
<td>25</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to short BFP)</td>
<td>LEXBR</td>
<td>RRE</td>
<td>Db Xi</td>
<td>B346</td>
</tr>
<tr>
<td>LOAD ROUNDED (extended to short BFP)</td>
<td>LEXR</td>
<td>RRE</td>
<td>Da EO</td>
<td>B366</td>
</tr>
<tr>
<td>LOAD ROUNDED (long to short BFP)</td>
<td>LEDBR</td>
<td>RRE</td>
<td>Db Xi</td>
<td>B344</td>
</tr>
<tr>
<td>LOAD ROUNDED (long to short HFP)</td>
<td>LEDR</td>
<td>RR</td>
<td>Da EO</td>
<td>35</td>
</tr>
<tr>
<td>LOAD ROUNDED (long to short HFP)</td>
<td>LDRR</td>
<td>RR</td>
<td>Da EO</td>
<td>35</td>
</tr>
<tr>
<td>LOAD USING REAL ADDRESS (32)</td>
<td>LURAG</td>
<td>RRE</td>
<td>Da EO</td>
<td>B24B</td>
</tr>
<tr>
<td>LOAD USING REAL ADDRESS (64)</td>
<td>LURAG</td>
<td>RRE</td>
<td>Da EO</td>
<td>B905</td>
</tr>
<tr>
<td>LOAD ZERO (extended)</td>
<td>LZXR</td>
<td>RRE</td>
<td>Da</td>
<td>B376</td>
</tr>
<tr>
<td>LOAD ZERO (long)</td>
<td>LZDR</td>
<td>RRE</td>
<td>Da</td>
<td>B375</td>
</tr>
<tr>
<td>LOAD ZERO (short)</td>
<td>LZR</td>
<td>RR</td>
<td>Da</td>
<td>B374</td>
</tr>
<tr>
<td>MODIFY STACKED STATE</td>
<td>MSTA</td>
<td>RRE</td>
<td>Da</td>
<td>B247</td>
</tr>
<tr>
<td>MODIFY SUBCHANNEL</td>
<td>MSCH</td>
<td>S C</td>
<td>Da</td>
<td>B232</td>
</tr>
<tr>
<td>MONITOR CALL</td>
<td>MC</td>
<td>SI</td>
<td>MO</td>
<td>AF</td>
</tr>
<tr>
<td>MOVE (character)</td>
<td>MVC</td>
<td>SS</td>
<td>A</td>
<td>ST B2 B2</td>
</tr>
<tr>
<td>MOVE (immediate)</td>
<td>MVI</td>
<td>SI</td>
<td>ST B1</td>
<td>92</td>
</tr>
<tr>
<td>MOVE (immediate)</td>
<td>MVII</td>
<td>SIY</td>
<td>ST B1</td>
<td>EB52</td>
</tr>
<tr>
<td>MOVE INVERSE</td>
<td>MVCIN</td>
<td>SS</td>
<td>ST B1 B2</td>
<td>E8</td>
</tr>
<tr>
<td>MOVE LONG</td>
<td>MVCL</td>
<td>RR C</td>
<td>II</td>
<td>ST R1 R2 O E</td>
</tr>
<tr>
<td>MOVE LONG EXTENDED</td>
<td>MVCLE</td>
<td>RS C</td>
<td>A SP</td>
<td>ST R1 R2 A</td>
</tr>
<tr>
<td>MOVE LONG UNICODE</td>
<td>MVCLU</td>
<td>RSY C E2</td>
<td>A SP</td>
<td>ST R1 R2 EBBE</td>
</tr>
<tr>
<td>MOVE NURMERIC</td>
<td>MVD</td>
<td>SS</td>
<td>ST B1</td>
<td>EB52</td>
</tr>
<tr>
<td>MOVE PAGE</td>
<td>MVPG</td>
<td>RRE C</td>
<td>ST B1 B2</td>
<td>E8</td>
</tr>
<tr>
<td>MOVE STRING</td>
<td>MVST</td>
<td>RRE</td>
<td>G0</td>
<td>ST R1 R2 B255</td>
</tr>
<tr>
<td>MOVE TO PRIMARY</td>
<td>MSCP</td>
<td>SS C</td>
<td>G0</td>
<td>ST R1 R2 B255</td>
</tr>
<tr>
<td>MOVE TO SECONDARY</td>
<td>MVS C</td>
<td>SS C</td>
<td>G0</td>
<td>ST R1 R2 B255</td>
</tr>
<tr>
<td>MOVE WITH DESTINATION KEY</td>
<td>MVCDK</td>
<td>SSE</td>
<td>GM</td>
<td>ST B1 B2 E50F</td>
</tr>
<tr>
<td>MOVE WITH KEY</td>
<td>MVCK</td>
<td>SS C</td>
<td>GM</td>
<td>B255</td>
</tr>
<tr>
<td>MOVE WITH OFFSET</td>
<td>MVO</td>
<td>SS</td>
<td>A</td>
<td>ST B1 B2 F1</td>
</tr>
<tr>
<td>MOVE WITH SOURCE KEY</td>
<td>MVCXK</td>
<td>SSE</td>
<td>G0</td>
<td>ST B1 B2 E50E</td>
</tr>
<tr>
<td>MOVE ZONES</td>
<td>MVZ</td>
<td>SS</td>
<td>A</td>
<td>ST B1 B2 D3</td>
</tr>
<tr>
<td>MULTIPLY (extended BFP)</td>
<td>MXBR</td>
<td>RRE</td>
<td>Db Xi</td>
<td>B34C</td>
</tr>
<tr>
<td>MULTIPLY (extended BFP)</td>
<td>MXR</td>
<td>RR</td>
<td>Da EU</td>
<td>26</td>
</tr>
<tr>
<td>MULTIPLY (long to extended BFP)</td>
<td>MXDBR</td>
<td>RRE</td>
<td>Db Xi</td>
<td>B307</td>
</tr>
<tr>
<td>MULTIPLY (long to extended BFP)</td>
<td>MXDB</td>
<td>RXE</td>
<td>Db Xi</td>
<td>B2</td>
</tr>
<tr>
<td>MULTIPLY (long to extended HFP)</td>
<td>MXDR</td>
<td>RR</td>
<td>Da EU</td>
<td>27</td>
</tr>
<tr>
<td>MULTIPLY (long to extended HFP)</td>
<td>MXD</td>
<td>RX</td>
<td>Da EU</td>
<td>67</td>
</tr>
<tr>
<td>MULTIPLY (long BFP)</td>
<td>MDRB</td>
<td>RRE</td>
<td>Db Xi</td>
<td>B31C</td>
</tr>
<tr>
<td>MULTIPLY (long BFP)</td>
<td>MDB</td>
<td>RXE</td>
<td>Db Xi</td>
<td>B2</td>
</tr>
<tr>
<td>MULTIPLY (long HFP)</td>
<td>MDR</td>
<td>RR</td>
<td>Da EU</td>
<td>B2</td>
</tr>
<tr>
<td>MULTIPLY (long HFP)</td>
<td>MD</td>
<td>RX</td>
<td>Da EU</td>
<td>2</td>
</tr>
<tr>
<td>MULTIPLY (short to long BFP)</td>
<td>MDEBR</td>
<td>RRE</td>
<td>Db Xi</td>
<td>B30C</td>
</tr>
<tr>
<td>MULTIPLY (short to long BFP)</td>
<td>MDEB</td>
<td>RXE</td>
<td>Db Xi</td>
<td>B2</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 9 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIPLY (short to long HFP)</td>
<td>MDER</td>
<td>RR</td>
<td>Da EU EO</td>
<td>3C</td>
</tr>
<tr>
<td>MULTIPLY (short to long HFP)</td>
<td>MER</td>
<td>RR</td>
<td>Da EU EO</td>
<td>3C</td>
</tr>
<tr>
<td>MULTIPLY (short to long HFP)</td>
<td>MDE</td>
<td>RX</td>
<td>A</td>
<td>7C</td>
</tr>
<tr>
<td>MULTIPLY (short to long HFP)</td>
<td>ME</td>
<td>RX</td>
<td>A</td>
<td>7C</td>
</tr>
<tr>
<td>MULTIPLY (short BFP)</td>
<td>MEEBR</td>
<td>RRE</td>
<td>Db Xi Xo Xu Xx</td>
<td>B317</td>
</tr>
<tr>
<td>MULTIPLY (short HFP)</td>
<td>MEER</td>
<td>RRE</td>
<td>Da EU EO</td>
<td>B337</td>
</tr>
<tr>
<td>MULTIPLY (short HFP)</td>
<td>MEEB</td>
<td>RXE</td>
<td>A</td>
<td>7C</td>
</tr>
<tr>
<td>MULTIPLY (short HFP)</td>
<td>ME</td>
<td>RXE</td>
<td>A</td>
<td>7C</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>MADBR</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B31E</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>MADBR</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B33E</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>MAEB</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B3/zerodotE</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short HFP)</td>
<td>MAE</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B3/zerodotE</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short HFP)</td>
<td>MAEB</td>
<td>RXF</td>
<td>A</td>
<td>7C</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short HFP)</td>
<td>MAEB</td>
<td>RXF</td>
<td>A</td>
<td>7C</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>MADB</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B31F</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>MADB</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B33F</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>MSB</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B3/zerodotF</td>
</tr>
<tr>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>MSB</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B3/zerodotF</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (long BFP)</td>
<td>MSDR</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (long BFP)</td>
<td>MSDR</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSBR</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSBR</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (long HFP)</td>
<td>MDR</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (long HFP)</td>
<td>MDR</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSB</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSB</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short HFP)</td>
<td>MSR</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY AND SUBTRACT (short HFP)</td>
<td>MSR</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>MULTIPLY DECIMAL</td>
<td>MP</td>
<td>SS</td>
<td>A</td>
<td>71</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD (32)</td>
<td>MH</td>
<td>RX</td>
<td>A</td>
<td>71</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD IMMEDIATE (32)</td>
<td>MHI</td>
<td>RI</td>
<td>A</td>
<td>71</td>
</tr>
<tr>
<td>MULTIPLY HALFWORD IMMEDIATE (64)</td>
<td>MGHI</td>
<td>RI</td>
<td>A</td>
<td>71</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>MLGR</td>
<td>RRE</td>
<td>SP</td>
<td>B986</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>MLG</td>
<td>RXY</td>
<td>N</td>
<td>B986</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>MLR</td>
<td>RRE</td>
<td>N3</td>
<td>B986</td>
</tr>
<tr>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>ML</td>
<td>RXY</td>
<td>N3</td>
<td>B986</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (32)</td>
<td>MS</td>
<td>RX</td>
<td>A</td>
<td>71</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (32)</td>
<td>MSY</td>
<td>RXY</td>
<td>A</td>
<td>71</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>MSGFR</td>
<td>RRE</td>
<td>N</td>
<td>96</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>MSGF</td>
<td>RXY</td>
<td>N</td>
<td>96</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (64)</td>
<td>MSGR</td>
<td>RRE</td>
<td>N</td>
<td>96</td>
</tr>
<tr>
<td>MULTIPLY SINGLE (64)</td>
<td>MSG</td>
<td>RXY</td>
<td>N</td>
<td>96</td>
</tr>
<tr>
<td>OR (character)</td>
<td>OC</td>
<td>SS</td>
<td>C</td>
<td>ST</td>
</tr>
<tr>
<td>OR (immediate)</td>
<td>OI</td>
<td>SI</td>
<td>C</td>
<td>ST</td>
</tr>
<tr>
<td>OR (immediate)</td>
<td>OIY</td>
<td>SIY</td>
<td>LD</td>
<td>ST</td>
</tr>
<tr>
<td>OR (32)</td>
<td>OR</td>
<td>RR</td>
<td>C</td>
<td>ST</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 10 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR (32)</td>
<td>OR</td>
<td>RX C A</td>
<td>B2</td>
<td>7-151</td>
</tr>
<tr>
<td>OR (32)</td>
<td>ORY</td>
<td>RXY C LD A</td>
<td>B2</td>
<td>7-152</td>
</tr>
<tr>
<td>OR (64)</td>
<td>ORG</td>
<td>RRE C N A</td>
<td>B2</td>
<td>7-151</td>
</tr>
<tr>
<td>OR (64)</td>
<td>OR</td>
<td>RXY C N A</td>
<td>B2</td>
<td>7-152</td>
</tr>
<tr>
<td>OR IMMEDIATE (high high)</td>
<td>ORIHH</td>
<td>RI C N</td>
<td>A5B</td>
<td>7-152</td>
</tr>
<tr>
<td>OR IMMEDIATE (low low)</td>
<td>ORIHL</td>
<td>RI C N</td>
<td>A5A</td>
<td>7-153</td>
</tr>
<tr>
<td>OR IMMEDIATE (low low)</td>
<td>ORIHL</td>
<td>RI C N</td>
<td>A5B</td>
<td>7-153</td>
</tr>
<tr>
<td>PACK</td>
<td>PACK</td>
<td>SS A</td>
<td>ST B1 B2</td>
<td>7-154</td>
</tr>
<tr>
<td>PACK ASCII</td>
<td>PKA</td>
<td>SS E2</td>
<td>ST B1 B2</td>
<td>7-155</td>
</tr>
<tr>
<td>PACK UNICODE</td>
<td>PKU</td>
<td>SS E2 A SP</td>
<td>ST B1 B2</td>
<td>7-155</td>
</tr>
<tr>
<td>PAGE IN</td>
<td>PGIN</td>
<td>RRE C ES P A1</td>
<td>B22E</td>
<td>7-156</td>
</tr>
<tr>
<td>PAGE OUT</td>
<td>PGOUT</td>
<td>RRE C ES P A1</td>
<td>B22F</td>
<td>7-156</td>
</tr>
<tr>
<td>PERFORMANCE LOCKED OPERATION</td>
<td>PLO</td>
<td>SS C A SP</td>
<td>ST FC</td>
<td>7-156</td>
</tr>
<tr>
<td>PROGRAM CALL</td>
<td>PC</td>
<td>S Q A1 Z1 T € GM</td>
<td>B ST</td>
<td>7-156</td>
</tr>
<tr>
<td>PROGRAM TRANSFER</td>
<td>PTI</td>
<td>RRE RA Q A1 A SP</td>
<td>B99E</td>
<td>7-156</td>
</tr>
<tr>
<td>PURGE ALB</td>
<td>PALB</td>
<td>RRE P $</td>
<td>B248</td>
<td>7-157</td>
</tr>
<tr>
<td>PURGE TLB</td>
<td>PTLB</td>
<td>S P $</td>
<td>B249</td>
<td>7-157</td>
</tr>
<tr>
<td>PROGRAM RETURN</td>
<td>PR</td>
<td>E L Q A1 A SP</td>
<td>B218</td>
<td>7-157</td>
</tr>
<tr>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>PT</td>
<td>RRE P Q A1 A SP</td>
<td>B219</td>
<td>7-157</td>
</tr>
<tr>
<td>RESUME PROGRAM</td>
<td>RP</td>
<td>S L Q A SP</td>
<td>B279</td>
<td>7-156</td>
</tr>
<tr>
<td>RESUME CHANNELED</td>
<td>RSCH</td>
<td>S C P</td>
<td>B277</td>
<td>7-157</td>
</tr>
<tr>
<td>SET ACCESS</td>
<td>SAL</td>
<td>S P OP $</td>
<td>B237</td>
<td>7-171</td>
</tr>
<tr>
<td>SET ADDRESS SPACE CONTROL</td>
<td>SAC</td>
<td>S Q SW €</td>
<td>B219</td>
<td>10-95</td>
</tr>
<tr>
<td>SET ADDRESS SPACE CONTROL FAST</td>
<td>SACF</td>
<td>S Q SW €</td>
<td>B279</td>
<td>10-95</td>
</tr>
<tr>
<td>SET ADDRESSING MODE (24)</td>
<td>SAM24</td>
<td>E N3 T</td>
<td>B279</td>
<td>10-95</td>
</tr>
<tr>
<td>SET ADDRESSING MODE (31)</td>
<td>SAM31</td>
<td>E N3 T</td>
<td>B279</td>
<td>10-95</td>
</tr>
<tr>
<td>SET ADDRESSING MODE (64)</td>
<td>SAM64</td>
<td>E N T</td>
<td>B279</td>
<td>10-95</td>
</tr>
<tr>
<td>SET CHANNEL MONITOR</td>
<td>SCHM</td>
<td>S P OP € GM</td>
<td>B237</td>
<td>14-12</td>
</tr>
<tr>
<td>SET CLOCK</td>
<td>SCK</td>
<td>S C A A SP</td>
<td>B218</td>
<td>14-12</td>
</tr>
<tr>
<td>SET CLOCK COMPARATOR</td>
<td>SCKC</td>
<td>S A A SP</td>
<td>B218</td>
<td>14-12</td>
</tr>
<tr>
<td>SET CLOCK PROGRAMMABLE FIELD</td>
<td>SCKPF</td>
<td>E A A SP</td>
<td>B218</td>
<td>14-12</td>
</tr>
<tr>
<td>SET CPU TIMER</td>
<td>SPT</td>
<td>S A A SP</td>
<td>B218</td>
<td>14-12</td>
</tr>
<tr>
<td>SET TCP</td>
<td>SFPC</td>
<td>RRE A Db</td>
<td>B234</td>
<td>19-43</td>
</tr>
<tr>
<td>SET PREFIX</td>
<td>SPX</td>
<td>S A $</td>
<td>B210</td>
<td>19-43</td>
</tr>
<tr>
<td>SET PROGRAM MASK</td>
<td>SPM</td>
<td>RR L</td>
<td>B20A</td>
<td>19-43</td>
</tr>
<tr>
<td>SET PSW KEY FROM ADDRESS</td>
<td>SPKA</td>
<td>S Q</td>
<td>B299</td>
<td>19-43</td>
</tr>
<tr>
<td>SET ROUNDING MODE</td>
<td>SRNM</td>
<td>S Db</td>
<td>B299</td>
<td>19-43</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 11 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET SECONDARY ASN</td>
<td>SSAR</td>
<td>RRE</td>
<td>A³</td>
<td>B225</td>
</tr>
<tr>
<td>SET SECONDARY ASN WITH INSTANCE</td>
<td>SSAIR</td>
<td>RRE RA</td>
<td>A³</td>
<td>B99F</td>
</tr>
<tr>
<td>SET STORAGE KEY EXTENDED</td>
<td>SSKE</td>
<td>RRE P A³ SP</td>
<td>SO</td>
<td>B22B</td>
</tr>
<tr>
<td>SET SYSTEM MASK</td>
<td>SSM</td>
<td>SS C</td>
<td>A</td>
<td>B0</td>
</tr>
<tr>
<td>SHIFT AND ROUND DECIMAL</td>
<td>SRP</td>
<td>SS C</td>
<td>A</td>
<td>F0</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE</td>
<td>SLDA</td>
<td>RS C</td>
<td>SP</td>
<td>B225</td>
</tr>
<tr>
<td>SHIFT LEFT DOUBLE LOGICAL</td>
<td>SLDL</td>
<td>RS C</td>
<td>SP</td>
<td>B99F</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE (32)</td>
<td>SLA</td>
<td>RS C</td>
<td>IF</td>
<td>B0</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE (64)</td>
<td>SLAG</td>
<td>RSY C N</td>
<td>IF</td>
<td>F0</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL (32)</td>
<td>SLL</td>
<td>RS</td>
<td>IF</td>
<td>F0</td>
</tr>
<tr>
<td>SHIFT LEFT SINGLE LOGICAL (64)</td>
<td>SLLG</td>
<td>RSY N</td>
<td>IF</td>
<td>B99F</td>
</tr>
<tr>
<td>SHIFT RIGHT DOUBLE</td>
<td>SRDA</td>
<td>RS C</td>
<td>IF</td>
<td>B0</td>
</tr>
<tr>
<td>SHIFT RIGHT DOUBLE LOGICAL</td>
<td>SRDL</td>
<td>RS C</td>
<td>IF</td>
<td>F0</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE (32)</td>
<td>SRA</td>
<td>RS C</td>
<td>IF</td>
<td>B0</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE (64)</td>
<td>SRAG</td>
<td>RSY C N</td>
<td>IF</td>
<td>F0</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE LOGICAL (32)</td>
<td>SRL</td>
<td>RS</td>
<td>IF</td>
<td>B0</td>
</tr>
<tr>
<td>SHIFT RIGHT SINGLE LOGICAL (64)</td>
<td>SRLG</td>
<td>RSY N</td>
<td>IF</td>
<td>F0</td>
</tr>
<tr>
<td>SIGNAL PROCESSOR</td>
<td>SIGP</td>
<td>RSY N P</td>
<td>S</td>
<td>AE</td>
</tr>
<tr>
<td>SQUARE ROOT (extended BFP)</td>
<td>SQXBR</td>
<td>RRE SP</td>
<td>Db X</td>
<td>B315</td>
</tr>
<tr>
<td>SQUARE ROOT (extended HFP)</td>
<td>SQXR</td>
<td>RRE SP</td>
<td>Da Sq</td>
<td>B316</td>
</tr>
<tr>
<td>SQUARE ROOT (long BFP)</td>
<td>SQDBR</td>
<td>RRE A</td>
<td>Db X</td>
<td>B315</td>
</tr>
<tr>
<td>SQUARE ROOT (long BFP)</td>
<td>SQDB</td>
<td>RXE A</td>
<td>Da Sq</td>
<td>B244</td>
</tr>
<tr>
<td>SQUARE ROOT (long HFP)</td>
<td>SQDR</td>
<td>RRE A</td>
<td>Db X</td>
<td>B244</td>
</tr>
<tr>
<td>SQUARE ROOT (long HFP)</td>
<td>SQD</td>
<td>RXE A</td>
<td>Da Sq</td>
<td>B244</td>
</tr>
<tr>
<td>SQUARE ROOT (short BFP)</td>
<td>SQEBR</td>
<td>RRE A</td>
<td>Db X</td>
<td>B315</td>
</tr>
<tr>
<td>SQUARE ROOT (short BFP)</td>
<td>SQEB</td>
<td>RXE A</td>
<td>Da Sq</td>
<td>B244</td>
</tr>
<tr>
<td>SQUARE ROOT (short HFP)</td>
<td>SQER</td>
<td>RRE A</td>
<td>Db X</td>
<td>B315</td>
</tr>
<tr>
<td>SQUARE ROOT (short HFP)</td>
<td>SQE</td>
<td>RXE A</td>
<td>Da Sq</td>
<td>B244</td>
</tr>
<tr>
<td>START SUBCHANNEL</td>
<td>SSCH</td>
<td>RX P A SP</td>
<td>Da GS</td>
<td>B233</td>
</tr>
<tr>
<td>STORE (long)</td>
<td>STDY</td>
<td>RX Y LD A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE (short)</td>
<td>STE</td>
<td>RX Y LD A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE (short)</td>
<td>STEY</td>
<td>RX Y LD A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE (32)</td>
<td>ST</td>
<td>RX Y LD A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE (32)</td>
<td>STY</td>
<td>RX Y LD A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE (64)</td>
<td>STG</td>
<td>RX Y N A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE ACCESS MULTIPLE</td>
<td>STAM</td>
<td>RX A SP</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE ACCESS MULTIPLE</td>
<td>STAMY</td>
<td>RSY LD A SP</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CHANNEL PATH STATUS</td>
<td>STCPS</td>
<td>S P A SP</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CHANNEL REPORT WORD</td>
<td>STCRM</td>
<td>S C P A SP</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CHARACTER</td>
<td>STC</td>
<td>RX A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CHARACTER</td>
<td>STCY</td>
<td>RX Y LD A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (high)</td>
<td>STCMH</td>
<td>RSY N A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>STCM</td>
<td>RX A</td>
<td>Da</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>STCMY</td>
<td>RSY LD A</td>
<td>Da</td>
<td>B2</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 12 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE CLOCK</td>
<td>STCK</td>
<td>S C A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CLOCK COMPARATOR</td>
<td>STCKC</td>
<td>S P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CLOCK EXTENDED</td>
<td>STCKE</td>
<td>S A $</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CONTROL (32)</td>
<td>STCTL</td>
<td>RS P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CONTROL (64)</td>
<td>STCTG</td>
<td>RSY N P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CPU ADDRESS</td>
<td>STAP</td>
<td>S P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CPU ID</td>
<td>STIDP</td>
<td>S P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE CPU TIMER</td>
<td>STPT</td>
<td>S P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE FACILITY LIST</td>
<td>STFL</td>
<td>S N3 P</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE FPC</td>
<td>STFPC</td>
<td>S A Db</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE HALFWORD</td>
<td>STH</td>
<td>RX A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE HALFWORD (32)</td>
<td>STHY</td>
<td>RXY LD A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE MULTIPLE (32)</td>
<td>STM</td>
<td>RS A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE MULTIPLE (64)</td>
<td>STMY</td>
<td>RSY LD A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE MULTIPLE HIGH</td>
<td>STMH</td>
<td>RSY N A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE PAIR TO QUADWORD</td>
<td>STPQ</td>
<td>RXY N A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE PREFIX</td>
<td>STPX</td>
<td>S P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE REAL ADDRESS</td>
<td>STRAG</td>
<td>SSE N P A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE REVISED (16)</td>
<td>STRV</td>
<td>RXY N3 A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE REVISED (32)</td>
<td>STRVG</td>
<td>RXY N A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE SUBCHANNEL</td>
<td>STSCH</td>
<td>S C P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE SYSTEM INFORMATION</td>
<td>STSI</td>
<td>S C P A SP</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE THEN AND SYSTEM MASK</td>
<td>STNRM</td>
<td>SI P A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE THEN OR SYSTEM MASK</td>
<td>STOSM</td>
<td>SI P A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>STORE USING REAL ADDRESS (32)</td>
<td>STURA</td>
<td>RRE P A SP £ GS</td>
<td>SU</td>
<td>B2</td>
</tr>
<tr>
<td>STORE USING REAL ADDRESS (64)</td>
<td>STURG</td>
<td>RRE N P A £ GS</td>
<td>SU</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (extended BFP)</td>
<td>SXBR</td>
<td>RRE C Db Xi Xo Xu Xx</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (long BFP)</td>
<td>SDBR</td>
<td>RRE C Db Xi Xo Xu Xx</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (long BFP)</td>
<td>SDB</td>
<td>RRE C A Db Xi Xo Xu Xx</td>
<td>B2</td>
<td>ED1B</td>
</tr>
<tr>
<td>SUBTRACT (short BFP)</td>
<td>SEB</td>
<td>RRE C A Db Xi Xo Xu Xx</td>
<td>B2</td>
<td>ED0B</td>
</tr>
<tr>
<td>SUBTRACT (short BFP)</td>
<td>SBE</td>
<td>RRE C A Db Xi Xo Xu Xx</td>
<td>B2</td>
<td>ED0B</td>
</tr>
<tr>
<td>SUBTRACT (32)</td>
<td>SR</td>
<td>RR C A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (32)</td>
<td>S</td>
<td>RX C A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (32)</td>
<td>SY</td>
<td>RXY C LD A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (64&lt;32)</td>
<td>SGFR</td>
<td>RRE C N A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (64&lt;32)</td>
<td>SGF</td>
<td>RXY C N A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (64)</td>
<td>SGR</td>
<td>RRE C N A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT (64)</td>
<td>SG</td>
<td>RXY C N A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT DECIMAL</td>
<td>SP</td>
<td>SS C A Dd DF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT HALFWORD</td>
<td>SH</td>
<td>RX C A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT HALFWORD</td>
<td>SHY</td>
<td>RXY C LD A IF</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (32)</td>
<td>SLR</td>
<td>RR C A IF</td>
<td>B2</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (32)</td>
<td>SL</td>
<td>RX C A IF</td>
<td>B2</td>
<td>B2</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 13 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBTRACT LOGICAL (32)</td>
<td>SLY</td>
<td>RXY C LD</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>SLGFR</td>
<td>RRE C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>SLGF</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64)</td>
<td>SLGR</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL (64)</td>
<td>SLG</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (32)</td>
<td>SLBR</td>
<td>RRE C N3</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (32)</td>
<td>SLB</td>
<td>RXY C N3</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (64)</td>
<td>SLBGR</td>
<td>RRE C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW (64)</td>
<td>SLBG</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED (extended HFP)</td>
<td>SDR</td>
<td>RRE C</td>
<td>Da EU EO LS</td>
<td>B1</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED (long HFP)</td>
<td>SDR</td>
<td>RRE C</td>
<td>Da EU EO LS</td>
<td>B1</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED (short HFP)</td>
<td>SER</td>
<td>RRE C</td>
<td>Da EU EO LS</td>
<td>B1</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED (short HFP)</td>
<td>SE</td>
<td>RRE C</td>
<td>Da EU EO LS</td>
<td>B1</td>
</tr>
<tr>
<td>SUBTRACT UNNORMALIZED (long HFP)</td>
<td>SWR</td>
<td>RR C</td>
<td>Da EU EO LS</td>
<td>B1</td>
</tr>
<tr>
<td>SUBTRACT UNNORMALIZED (short HFP)</td>
<td>SWR</td>
<td>RR C</td>
<td>Da EU EO LS</td>
<td>B1</td>
</tr>
<tr>
<td>SUPERVISOR CALL</td>
<td>SVC</td>
<td>I</td>
<td>$</td>
<td>ST</td>
</tr>
<tr>
<td>TEST ADDRESSING MODE</td>
<td>TAM</td>
<td>E C N3</td>
<td>A</td>
<td>ST</td>
</tr>
<tr>
<td>TEST AND SET</td>
<td>TS</td>
<td>S C</td>
<td>A</td>
<td>ST</td>
</tr>
<tr>
<td>TEST BLOCK</td>
<td>TB</td>
<td>RRE C P A1</td>
<td>II $ G0</td>
<td>ST</td>
</tr>
<tr>
<td>TEST DATA CLASS (extended BFP)</td>
<td>TCB</td>
<td>RXE C SP</td>
<td>Db</td>
<td>ST</td>
</tr>
<tr>
<td>TEST DATA CLASS (long BFP)</td>
<td>TDB</td>
<td>RXE C</td>
<td>Db</td>
<td>ST</td>
</tr>
<tr>
<td>TEST DATA CLASS (short BFP)</td>
<td>TCEB</td>
<td>RXE C A</td>
<td>Db</td>
<td>ST</td>
</tr>
<tr>
<td>TEST DECIMAL</td>
<td>TP</td>
<td>RSL C E2</td>
<td>A</td>
<td>ST</td>
</tr>
<tr>
<td>TEST PENDING INTERRUPTION</td>
<td>TPI</td>
<td>S C P A1 SP</td>
<td>$ GS</td>
<td>ST</td>
</tr>
<tr>
<td>TEST PROTECTION</td>
<td>TPROT</td>
<td>SSE C P A1</td>
<td>$ GS</td>
<td>ST</td>
</tr>
<tr>
<td>TEST SUBCHANNEL</td>
<td>TSCH</td>
<td>S C P A SP</td>
<td>OP $ GS</td>
<td>ST</td>
</tr>
<tr>
<td>TEST UNDER MASK</td>
<td>TM</td>
<td>SI C A</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TEST UNDER MASK</td>
<td>TMY</td>
<td>SIY C LD</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TEST UNDER MASK (high high)</td>
<td>TMHH</td>
<td>RI C N</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TEST UNDER MASK (high low)</td>
<td>TMHL</td>
<td>RI C N</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TEST UNDER MASK (low high)</td>
<td>TMLH</td>
<td>RI C N</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TEST UNDER MASK (low low)</td>
<td>TMLL</td>
<td>RI C N</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TEST UNDER MASK HIGH</td>
<td>TMLL</td>
<td>RI C N</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TEST UNDER MASK LOW</td>
<td>TML</td>
<td>RI C</td>
<td>A</td>
<td>B1</td>
</tr>
<tr>
<td>TRACE (32)</td>
<td>TRACE</td>
<td>RS P A SP T $</td>
<td>99</td>
<td>B2</td>
</tr>
<tr>
<td>TRACE (64)</td>
<td>TRACG</td>
<td>RSY N P A SP T $</td>
<td>99</td>
<td>B2</td>
</tr>
<tr>
<td>TRANSLATE</td>
<td>TR</td>
<td>SS A</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>TRANSLATE AND TEST</td>
<td>TRT</td>
<td>SS C A</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>TRANSLATE AND TEST REVERSED</td>
<td>TRTR</td>
<td>SS C E3</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>TRANSLATE EXTENDED</td>
<td>TRE</td>
<td>RRE C A</td>
<td>ST</td>
<td></td>
</tr>
<tr>
<td>TRANSLATE ONE TO ONE</td>
<td>TROO</td>
<td>RRE C E2 A SP</td>
<td>ST</td>
<td></td>
</tr>
</tbody>
</table>

Figure B-1 (Part 14 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSLATE ONE TO TWO</td>
<td>TROT</td>
<td>RRE C E2</td>
<td>A SP</td>
<td>GM</td>
</tr>
<tr>
<td>TRANSLATE TWO TO ONE</td>
<td>TRTO</td>
<td>RRE C E2</td>
<td>A SP</td>
<td>GM</td>
</tr>
<tr>
<td>TRANSLATE TWO TO TWO</td>
<td>TRTT</td>
<td>RRE C E2</td>
<td>A SP</td>
<td>GM</td>
</tr>
<tr>
<td>TRAP</td>
<td>TRAP2</td>
<td>E</td>
<td>A</td>
<td>SO T</td>
</tr>
<tr>
<td>TRAP</td>
<td>TRAP4</td>
<td>S</td>
<td>A</td>
<td>SO T</td>
</tr>
<tr>
<td>UNPACK</td>
<td>UNPK</td>
<td>SS</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>UNPACK ASCII</td>
<td>UNPKA</td>
<td>SS C E2</td>
<td>A SP</td>
<td></td>
</tr>
<tr>
<td>UNPACK UNICODE</td>
<td>UNPKU</td>
<td>SS C E2</td>
<td>A SP</td>
<td></td>
</tr>
<tr>
<td>UPDATE TREE</td>
<td>UPT</td>
<td>E C</td>
<td>A SP</td>
<td>II GM</td>
</tr>
<tr>
<td>ZERO AND ADD</td>
<td>ZAP</td>
<td>SS C</td>
<td>A</td>
<td>Dd DF</td>
</tr>
</tbody>
</table>

Figure B-1 (Part 15 of 15). Instructions Arranged by Name
<table>
<thead>
<tr>
<th>Mne- monic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A DIAGNOSE</td>
<td>RX C</td>
<td>IF</td>
<td>B3</td>
<td>10-19</td>
</tr>
<tr>
<td>ADD (32)</td>
<td>RX C</td>
<td>Da EU EO LS</td>
<td>B2</td>
<td>7-18</td>
</tr>
<tr>
<td>AD ADD NORMALIZED (long HFP)</td>
<td>RX C</td>
<td>Db Xi Xu Xx</td>
<td>B2</td>
<td>18-8</td>
</tr>
<tr>
<td>ADB ADD (long BFP)</td>
<td>RXE C</td>
<td>Db Xi Xu Xx</td>
<td>B2</td>
<td>19-18</td>
</tr>
<tr>
<td>ADBR ADD (long BFP)</td>
<td>RRE C</td>
<td>Da EU EO LS</td>
<td>B2</td>
<td>19-18</td>
</tr>
<tr>
<td>ADR ADD NORMALIZED (long HFP)</td>
<td>RX C</td>
<td>Db Xi Xu Xx</td>
<td>B2</td>
<td>18-8</td>
</tr>
<tr>
<td>AE ADD NORMALIZED (short HFP)</td>
<td>RX C</td>
<td>Da EU EO LS</td>
<td>B2</td>
<td>18-8</td>
</tr>
<tr>
<td>AEB ADD (short BFP)</td>
<td>RXE C</td>
<td>Db Xi Xu Xx</td>
<td>B2</td>
<td>19-18</td>
</tr>
<tr>
<td>AEBR ADD (short BFP)</td>
<td>RRE C</td>
<td>Db Xi Xu Xx</td>
<td>B2</td>
<td>19-18</td>
</tr>
<tr>
<td>AER ADD NORMALIZED (short HFP)</td>
<td>RR C</td>
<td>Da EU EO LS</td>
<td>B2</td>
<td>3A</td>
</tr>
<tr>
<td>AG ADD (64)</td>
<td>RXY C N</td>
<td>A</td>
<td>E3</td>
<td>7-18</td>
</tr>
<tr>
<td>AGF ADD (64&lt;32)</td>
<td>RXY C N</td>
<td>A</td>
<td>E318</td>
<td>7-18</td>
</tr>
<tr>
<td>AGFR ADD HALFWORD IMMEDIATE (64)</td>
<td>RRE C N</td>
<td>IF</td>
<td>B9</td>
<td>7-18</td>
</tr>
<tr>
<td>AGR ADD (64)</td>
<td>RR C</td>
<td>IF</td>
<td>B9/zerodot8</td>
<td>7-18</td>
</tr>
<tr>
<td>AH ADD HALFWORD</td>
<td>RX C</td>
<td>IF</td>
<td>E37A</td>
<td>7-18</td>
</tr>
<tr>
<td>AHI ADD HALFWORD IMMEDIATE (32)</td>
<td>RX C</td>
<td>IF</td>
<td>E398</td>
<td>7-19</td>
</tr>
<tr>
<td>AHY ADD HALFWORD</td>
<td>RXY C LD</td>
<td>A</td>
<td>E398</td>
<td>7-18</td>
</tr>
<tr>
<td>AH C ADD LOGICAL (32)</td>
<td>RXY C N3</td>
<td>A</td>
<td>E398</td>
<td>7-18</td>
</tr>
<tr>
<td>ALC ADD LOGICAL WITH CARRY (32)</td>
<td>RXY C N</td>
<td>A</td>
<td>E398</td>
<td>7-20</td>
</tr>
<tr>
<td>ALCG ADD LOGICAL WITH CARRY (64)</td>
<td>RXY C N</td>
<td>A</td>
<td>B918</td>
<td>7-19</td>
</tr>
<tr>
<td>ALCGR ADD LOGICAL WITH CARRY (64)</td>
<td>RRE C N</td>
<td>A</td>
<td>E398</td>
<td>7-19</td>
</tr>
<tr>
<td>ALCR ADD LOGICAL WITH CARRY (32)</td>
<td>RR C</td>
<td>A</td>
<td>E398</td>
<td>7-19</td>
</tr>
<tr>
<td>ALC崧 ADD LOGICAL (32)</td>
<td>RXY C N</td>
<td>A</td>
<td>E398</td>
<td>7-19</td>
</tr>
<tr>
<td>ALCGFR ADD LOGICAL (64&lt;32)</td>
<td>RXY C N</td>
<td>IF</td>
<td>B91A</td>
<td>7-19</td>
</tr>
<tr>
<td>ALGR ADD LOGICAL (64)</td>
<td>RXY C N</td>
<td>IF</td>
<td>B90A</td>
<td>7-19</td>
</tr>
<tr>
<td>ALR ADD LOGICAL (32)</td>
<td>RXY C LD</td>
<td>IF</td>
<td>B90A</td>
<td>7-19</td>
</tr>
<tr>
<td>ALC崧 ADD LOGICAL WITH CARRY (32)</td>
<td>RR C</td>
<td>IF</td>
<td>B90A</td>
<td>7-19</td>
</tr>
<tr>
<td>AP ADD DECIMAL</td>
<td>SS C</td>
<td>Da DF</td>
<td>FA</td>
<td>8-5</td>
</tr>
</tbody>
</table>

**Figure B-2 (Part 1 of 15). Instructions Arranged by Mnemonic**
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCR</td>
<td>BRANCH ON CONDITION</td>
<td>RR</td>
<td>(\xi^1)</td>
<td>B</td>
</tr>
<tr>
<td>BCT</td>
<td>BRANCH ON COUNT (32)</td>
<td>RX</td>
<td>B</td>
<td>46</td>
</tr>
<tr>
<td>BCTG</td>
<td>BRANCH ON COUNT (64)</td>
<td>RXY N</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BCTR</td>
<td>BRANCH ON COUNT (64)</td>
<td>RRE N</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BRAS</td>
<td>BRANCH RELATIVE AND SAVE</td>
<td>RI</td>
<td>B</td>
<td>A75</td>
</tr>
<tr>
<td>BRASL</td>
<td>BRANCH RELATIVE AND SAVE LONG</td>
<td>RIL N3</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BRC</td>
<td>BRANCH RELATIVE ON CONDITION</td>
<td>RI</td>
<td>B</td>
<td>A74</td>
</tr>
<tr>
<td>BRCL</td>
<td>BRANCH RELATIVE ON CONDITION LONG</td>
<td>RIL N3</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BCTR</td>
<td>BRANCH RELATIVE ON COUNT (32)</td>
<td>RI</td>
<td>B</td>
<td>A76</td>
</tr>
<tr>
<td>BRAS</td>
<td>BRANCH RELATIVE AND SAVE</td>
<td>RI</td>
<td>B</td>
<td>A75</td>
</tr>
<tr>
<td>BRASL</td>
<td>BRANCH RELATIVE AND SAVE LONG</td>
<td>RIL N3</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BRC</td>
<td>BRANCH RELATIVE ON CONDITION</td>
<td>RI</td>
<td>B</td>
<td>A74</td>
</tr>
<tr>
<td>BRCL</td>
<td>BRANCH RELATIVE ON CONDITION LONG</td>
<td>RIL N3</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BCTR</td>
<td>BRANCH RELATIVE ON COUNT (32)</td>
<td>RI</td>
<td>B</td>
<td>A76</td>
</tr>
<tr>
<td>BRAS</td>
<td>BRANCH RELATIVE AND SAVE</td>
<td>RI</td>
<td>B</td>
<td>A75</td>
</tr>
<tr>
<td>BRASL</td>
<td>BRANCH RELATIVE AND SAVE LONG</td>
<td>RIL N3</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BRC</td>
<td>BRANCH RELATIVE ON CONDITION</td>
<td>RI</td>
<td>B</td>
<td>A74</td>
</tr>
<tr>
<td>BRCL</td>
<td>BRANCH RELATIVE ON CONDITION LONG</td>
<td>RIL N3</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>BCTR</td>
<td>BRANCH RELATIVE ON COUNT (32)</td>
<td>RI</td>
<td>B</td>
<td>A76</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 2 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFEBR</td>
<td>CONVERT TO FIXED (short BFP to 32)</td>
<td>RRF C</td>
<td>Db Xi</td>
<td>B398 19-26</td>
</tr>
<tr>
<td>CFER</td>
<td>CONVERT TO FIXED (short HFP to 32)</td>
<td>RRF C</td>
<td>Da</td>
<td>B380 18-11</td>
</tr>
<tr>
<td>CFXBR</td>
<td>CONVERT TO FIXED (ext. BFP to 32)</td>
<td>RRF C</td>
<td>Db Xi</td>
<td>B39A 19-26</td>
</tr>
<tr>
<td>CFXR</td>
<td>CONVERT TO FIXED (ext. HFP to 32)</td>
<td>RRF C</td>
<td>Da</td>
<td>B38A 18-11</td>
</tr>
<tr>
<td>CGDBR</td>
<td>CONVERT TO FIXED (long BFP to 64)</td>
<td>RRF C N</td>
<td>Db Xi</td>
<td>B39C 19-26</td>
</tr>
<tr>
<td>CGDR</td>
<td>CONVERT TO FIXED (long HFP to 64)</td>
<td>RRF C N</td>
<td>Da</td>
<td>B38B 18-11</td>
</tr>
<tr>
<td>CGEBR</td>
<td>CONVERT TO FIXED (short BFP to 64)</td>
<td>RRF C N</td>
<td>Db Xi</td>
<td>B39D 19-26</td>
</tr>
<tr>
<td>CGER</td>
<td>CONVERT TO FIXED (short HFP to 64)</td>
<td>RRF C N</td>
<td>Da</td>
<td>B38C 18-11</td>
</tr>
<tr>
<td>CGF</td>
<td>COMPARE (64&lt;32)</td>
<td>RXY C N</td>
<td>Db Xi</td>
<td>B39E 19-26</td>
</tr>
<tr>
<td>CGFR</td>
<td>COMPARE (64&lt;32)</td>
<td>RRE C N</td>
<td>Da</td>
<td>B38D 18-11</td>
</tr>
<tr>
<td>CGH</td>
<td>COMPARE HALFWORD</td>
<td>RX C</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CHI</td>
<td>COMPARE HALFWORD IMMEDIATE (32)</td>
<td>RX Y C L D</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CHY</td>
<td>COMPARE HALFWORD</td>
<td>RX Y C L D</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CKS M</td>
<td>CHECKSUM</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CL</td>
<td>COMPARE LOGICAL (32)</td>
<td>RX C</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLC</td>
<td>COMPARE LOGICAL (character)</td>
<td>SS C</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLCC</td>
<td>COMPARE LOGICAL LONG</td>
<td>RX C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLCL</td>
<td>COMPARE LOGICAL LONG EXTENDED</td>
<td>RX C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLCLU</td>
<td>COMPARE LOGICAL LONG UNICODE</td>
<td>RX C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLG</td>
<td>COMPARE LOGICAL (64)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLGF</td>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLGFR</td>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLGR</td>
<td>COMPARE LOGICAL (64)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLI</td>
<td>COMPARE LOGICAL (immediate)</td>
<td>SI C</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLY</td>
<td>COMPARE LOGICAL (32)</td>
<td>RX Y C L D</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLMY</td>
<td>COMPARE LOGICAL (32)</td>
<td>RX Y C L D</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLR</td>
<td>COMPARE LOGICAL (32)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLST</td>
<td>COMPARE LOGICAL STRING</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CLY</td>
<td>COMPARE LOGICAL (32)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CMPSC</td>
<td>COMPRESSION CALL</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CP</td>
<td>COMPARE DECIMAL</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CPYA</td>
<td>COPY ACCESS</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CR</td>
<td>COMPARE (32)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CS</td>
<td>COMPARE AND SWAP (32)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CSCH</td>
<td>CLEAR SUBCHANNEL</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CSG</td>
<td>COMPARE AND SWAP (64)</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CSP</td>
<td>COMPARE AND SWAP AND PURGE</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
<tr>
<td>CSPG</td>
<td>COMPARE AND SWAP AND PURGE</td>
<td>RX Y C N</td>
<td>A</td>
<td>B24 7-53</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 3 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSY</td>
<td>COMPARE AND SWAP (32)</td>
<td>RSY C LD A SP $</td>
<td>ST B2</td>
<td>EB14 7-53</td>
</tr>
<tr>
<td>CUSE</td>
<td>COMPARE UNTIL SUBSTRING EQUAL</td>
<td>RRE C A SP II GM</td>
<td>RT RS B2 R2</td>
<td>B257 7-68</td>
</tr>
<tr>
<td>CUTFU</td>
<td>CONVERT UTF-8 TO UNICODE</td>
<td>RRE C A SP</td>
<td>ST RS R2 R2</td>
<td>B2A7 7-101</td>
</tr>
<tr>
<td>CUUTF</td>
<td>CONVERT UNICODE TO UTF-8</td>
<td>RRE C A SP</td>
<td>ST RS R2 R2</td>
<td>B2A6 7-101</td>
</tr>
<tr>
<td>CU12</td>
<td>CONVERT UTF-8 TO UTF-16</td>
<td>RRE C E3 A SP</td>
<td>ST RS R2 R2</td>
<td>B2A7 7-101</td>
</tr>
<tr>
<td>CUSE</td>
<td>COMPARE UNTIL SUBSTRING EQUAL</td>
<td>RRE C A SP II GM</td>
<td>RT RS B2 R2</td>
<td>B2A7 7-101</td>
</tr>
<tr>
<td>CUTFU</td>
<td>CONVERT UTF-8 TO UTF-16</td>
<td>RRE C E3 A SP</td>
<td>ST RS R2 R2</td>
<td>B2A6 7-101</td>
</tr>
<tr>
<td>CUTE</td>
<td>CONVERT UTF-32 TO UTF-8</td>
<td>RRE C E3 A SP</td>
<td>ST RS R2 R2</td>
<td>B2A6 7-101</td>
</tr>
<tr>
<td>CUTFU</td>
<td>CONVERT UTF-32 TO UTF-16</td>
<td>RRE C E3 A SP</td>
<td>ST RS R2 R2</td>
<td>B2A7 7-101</td>
</tr>
<tr>
<td>CVB</td>
<td>CONVERT TO BINARY (32)</td>
<td>RX A Dd IK</td>
<td>B B2</td>
<td>4F 7-97</td>
</tr>
<tr>
<td>CVBG</td>
<td>CONVERT TO BINARY (64)</td>
<td>RXY N A Dd IK</td>
<td>B B2</td>
<td>E32E 7-97</td>
</tr>
<tr>
<td>CVBY</td>
<td>CONVERT TO BINARY (32)</td>
<td>RXY LD A Dd IK</td>
<td>B B2</td>
<td>E32E 7-97</td>
</tr>
<tr>
<td>CVD</td>
<td>CONVERT TO DECIMAL (32)</td>
<td>RX A</td>
<td>ST B2</td>
<td>4E 7-98</td>
</tr>
<tr>
<td>CVGD</td>
<td>CONVERT TO DECIMAL (64)</td>
<td>RXY N A</td>
<td>ST B2</td>
<td>E32E 7-98</td>
</tr>
<tr>
<td>CVDY</td>
<td>CONVERT TO DECIMAL (32)</td>
<td>RXY LD A Db Xi</td>
<td>ST B2</td>
<td>E326 7-98</td>
</tr>
<tr>
<td>CBXBR</td>
<td>COMPARE (extended BFP)</td>
<td>RRE C A Db Xi Xz Xo Xu Xx</td>
<td>ST B2</td>
<td>E326 7-98</td>
</tr>
<tr>
<td>CXFBR</td>
<td>CONVERT FROM FIXED (32 to ext. BFP)</td>
<td>RRE SP Db</td>
<td>B2</td>
<td>E359 19-23</td>
</tr>
<tr>
<td>CXFR</td>
<td>CONVERT FROM FIXED (32 to ext. HFP)</td>
<td>RRE SP Da</td>
<td>B2</td>
<td>E359 19-23</td>
</tr>
<tr>
<td>CXGBR</td>
<td>CONVERT FROM FIXED (64 to ext. BFP)</td>
<td>RRE N SP Db</td>
<td>B2</td>
<td>B349 19-23</td>
</tr>
<tr>
<td>CXGR</td>
<td>CONVERT FROM FIXED (64 to ext. HFP)</td>
<td>RRE N SP Da</td>
<td>B2</td>
<td>4F 7-97</td>
</tr>
<tr>
<td>CXR</td>
<td>COMPARE (extended HFP)</td>
<td>RRE C A Da</td>
<td>ST B2</td>
<td>4E 7-98</td>
</tr>
<tr>
<td>CY</td>
<td>COMPARE (32)</td>
<td>RXY C LD A Sp IK</td>
<td>B2</td>
<td>4E 7-98</td>
</tr>
<tr>
<td>D</td>
<td>DIVIDE (32&lt;64)</td>
<td>RX A Da EU EO FK</td>
<td>B2</td>
<td>5D 18-12</td>
</tr>
<tr>
<td>DD</td>
<td>DIVIDE (long HFP)</td>
<td>RX A Da EU EO FK</td>
<td>B2</td>
<td>5D 18-12</td>
</tr>
<tr>
<td>DDB</td>
<td>DIVIDE (long BFP)</td>
<td>RXE A Db Xi Xz Xo Xu Xx</td>
<td>B2</td>
<td>E326 7-98</td>
</tr>
<tr>
<td>DDBR</td>
<td>DIVIDE (long BFP)</td>
<td>RRE A Db Xi Xz Xo Xu Xx</td>
<td>B2</td>
<td>B31D 19-26</td>
</tr>
<tr>
<td>DDR</td>
<td>DIVIDE (long HFP)</td>
<td>RX A Da EU EO FK</td>
<td>B2</td>
<td>7D 18-12</td>
</tr>
<tr>
<td>DE</td>
<td>DIVIDE (short HFP)</td>
<td>RX A Da EU EO FK</td>
<td>B2</td>
<td>E397 7-117</td>
</tr>
<tr>
<td>DEB</td>
<td>DIVIDE (short BFP)</td>
<td>RXE A Db Xi Xz Xo Xu Xx</td>
<td>B2</td>
<td>E397 7-117</td>
</tr>
<tr>
<td>DEBR</td>
<td>DIVIDE (short BFP)</td>
<td>RRF C A Db Xi Xz Xo Xu Xx</td>
<td>B2</td>
<td>30 18-12</td>
</tr>
<tr>
<td>DER</td>
<td>DIVIDE (short HFP)</td>
<td>RR Da EU EO FK</td>
<td>B2</td>
<td>30 18-12</td>
</tr>
<tr>
<td>DIDBR</td>
<td>DIVIDE TO INTEGER (long BFP)</td>
<td>RRF C A Db Xi Xz Xo Xu Xx</td>
<td>B2</td>
<td>E353 19-29</td>
</tr>
<tr>
<td>DIEBR</td>
<td>DIVIDE TO INTEGER (short BFP)</td>
<td>RRF C A Db Xi Xz Xo Xu Xx</td>
<td>B2</td>
<td>E353 19-29</td>
</tr>
<tr>
<td>DL</td>
<td>DIVIDE LOGICAL (32&lt;64)</td>
<td>RXY N3 A SP IK</td>
<td>B2</td>
<td>E397 7-117</td>
</tr>
<tr>
<td>DLG</td>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>RXY N A SP IK</td>
<td>B2</td>
<td>E387 7-117</td>
</tr>
<tr>
<td>DLR</td>
<td>DIVIDE LOGICAL (32&lt;64)</td>
<td>RRF N A SP IK</td>
<td>B2</td>
<td>E387 7-117</td>
</tr>
<tr>
<td>DLGR</td>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>RRF N A SP IK</td>
<td>B2</td>
<td>E387 7-117</td>
</tr>
<tr>
<td>DP</td>
<td>DIVIDE DECIMAL</td>
<td>SS A SP Dd DK</td>
<td>ST B1 B2</td>
<td>FD 8-6</td>
</tr>
<tr>
<td>DR</td>
<td>DIVIDE (32&lt;64)</td>
<td>RR A SP IK</td>
<td>ST B1 B2</td>
<td>FD 8-6</td>
</tr>
<tr>
<td>DSG</td>
<td>DIVIDE SINGLE (64)</td>
<td>RXY N A SP IK</td>
<td>B2</td>
<td>E30D 7-117</td>
</tr>
<tr>
<td>DSGF</td>
<td>DIVIDE SINGLE (64&lt;32)</td>
<td>RXY N A SP IK</td>
<td>B2</td>
<td>E31D 7-117</td>
</tr>
<tr>
<td>DSGFR</td>
<td>DIVIDE SINGLE (64&lt;32)</td>
<td>RXY N A SP IK</td>
<td>B2</td>
<td>E31D 7-117</td>
</tr>
<tr>
<td>DSGR</td>
<td>DIVIDE SINGLE (64)</td>
<td>RXY N A SP IK</td>
<td>B2</td>
<td>E30D 7-117</td>
</tr>
<tr>
<td>DXBR</td>
<td>DIVIDE (extended BFP)</td>
<td>RRE A Db Xi Xz Xo Xu Xx</td>
<td>B2</td>
<td>B34D 19-29</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 4 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXR</td>
<td>DIVIDE (extended HFP)</td>
<td>RRE</td>
<td>SP</td>
<td>Da EU EO FK</td>
</tr>
<tr>
<td>EAR</td>
<td>EXTRACT ACCESS</td>
<td>RRE</td>
<td>SP</td>
<td>Da EU EO FK</td>
</tr>
<tr>
<td>ED</td>
<td>EDIT</td>
<td>SS C</td>
<td>A</td>
<td>Dd</td>
</tr>
<tr>
<td>EDMK</td>
<td>EDIT AND MARK</td>
<td>SS C</td>
<td>A</td>
<td>Dd</td>
</tr>
<tr>
<td>EFPC</td>
<td>EXTRACT FPC</td>
<td>RRE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPAIR</td>
<td>EXTRACT PRIMARY ASN AND INSTANCE</td>
<td>RRE RA</td>
<td>Q</td>
<td>SO</td>
</tr>
<tr>
<td>EPAR</td>
<td>EXTRACT PRIMARY ASN</td>
<td>RRE</td>
<td>Q</td>
<td>SO</td>
</tr>
<tr>
<td>EPSW</td>
<td>EXTRACT PSW</td>
<td>RRE N3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EREG</td>
<td>EXTRACT STACKED REGISTERS (32)</td>
<td>RRE N</td>
<td>A</td>
<td>SE</td>
</tr>
<tr>
<td>EREGG</td>
<td>EXTRACT STACKED REGISTERS (64)</td>
<td>RRE N</td>
<td>A</td>
<td>SE</td>
</tr>
<tr>
<td>EPAIR</td>
<td>EXTRACT SECONDARY ASN AND INSTANCE</td>
<td>RRE RA</td>
<td>Q</td>
<td>SO</td>
</tr>
<tr>
<td>ESAR</td>
<td>EXTRACT SECONDARY ASN</td>
<td>RRE</td>
<td>Q</td>
<td>SO</td>
</tr>
<tr>
<td>ESEA</td>
<td>EXTRACT AND SET EXTENDED AUTHORITY</td>
<td>RRE C</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>EXECUTE</td>
<td>RX</td>
<td>AI SP</td>
<td>EX</td>
</tr>
<tr>
<td>FIDBR</td>
<td>LOAD FP INTEGER (long BFP)</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>FIDR</td>
<td>LOAD FP INTEGER (long HFP)</td>
<td>RRE</td>
<td>SP</td>
<td>Da</td>
</tr>
<tr>
<td>FIEBR</td>
<td>LOAD FP INTEGER (short BFP)</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>FIER</td>
<td>LOAD FP INTEGER (short HFP)</td>
<td>RRE</td>
<td>SP</td>
<td>Da</td>
</tr>
<tr>
<td>FIXBR</td>
<td>LOAD FP INTEGER (extended BFP)</td>
<td>RRF</td>
<td>SP</td>
<td>Db Xi Xx</td>
</tr>
<tr>
<td>FIXR</td>
<td>LOAD FP INTEGER (extended HFP)</td>
<td>RRE</td>
<td>SP</td>
<td>Da</td>
</tr>
<tr>
<td>HDR</td>
<td>HALVE (long HFP)</td>
<td>RR</td>
<td>SP</td>
<td>Da EU</td>
</tr>
<tr>
<td>HER</td>
<td>HALVE (short HFP)</td>
<td>RR</td>
<td>SP</td>
<td>Da EU</td>
</tr>
<tr>
<td>HSCH</td>
<td>HALT SUBCHANNEL</td>
<td>S C</td>
<td>P</td>
<td>OP &amp; GS</td>
</tr>
<tr>
<td>IAC</td>
<td>INSERT ADDRESS SPACE CONTROL</td>
<td>RRE C</td>
<td>Q</td>
<td>SO</td>
</tr>
<tr>
<td>IC</td>
<td>INSERT CHARACTER</td>
<td>RX</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ICM</td>
<td>INSERT CHARACTERS UNDER MASK (low)</td>
<td>RS C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ICMD</td>
<td>INSERT CHARACTERS UNDER MASK (high)</td>
<td>RSY C N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ICMT</td>
<td>INSERT CHARACTERS UNDER MASK (low)</td>
<td>RSY C LD</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ICMTY</td>
<td>INSERT CHARACTERS UNDER MASK (low)</td>
<td>RSY C LD</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>ICMTY</td>
<td>INSERT CHARACTER</td>
<td>RX Y LD</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>IDTE</td>
<td>INVALIDATE DAT TABLE ENTRY</td>
<td>RRF DE</td>
<td>P A</td>
<td>$</td>
</tr>
<tr>
<td>IHH</td>
<td>INSERT IMMEDIATE (high high)</td>
<td>RI N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IIHL</td>
<td>INSERT IMMEDIATE (high low)</td>
<td>RI N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILLH</td>
<td>INSERT IMMEDIATE (low high)</td>
<td>RI N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IILL</td>
<td>INSERT IMMEDIATE (low low)</td>
<td>RI N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPK</td>
<td>INSERT PSW KEY</td>
<td>S</td>
<td>Q</td>
<td>G2</td>
</tr>
<tr>
<td>IPM</td>
<td>INSERT PROGRAM MASK</td>
<td>RRE</td>
<td>P A</td>
<td>$</td>
</tr>
<tr>
<td>IPTE</td>
<td>INVALIDATE PAGE TABLE ENTRY</td>
<td>RRE</td>
<td>P A</td>
<td>$</td>
</tr>
<tr>
<td>ISKE</td>
<td>INSERT STORAGE KEY EXTENDED</td>
<td>RRE</td>
<td>P A</td>
<td>$</td>
</tr>
<tr>
<td>IVSK</td>
<td>INSERT VIRTUAL STORAGE KEY</td>
<td>RRE</td>
<td>Q A</td>
<td>SO</td>
</tr>
<tr>
<td>KDB</td>
<td>COMPARE AND SIGNAL (long BFP)</td>
<td>RXE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>KDBR</td>
<td>COMPARE AND SIGNAL (long BFP)</td>
<td>RXE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>KEB</td>
<td>COMPARE AND SIGNAL (short BFP)</td>
<td>RXE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>KEBR</td>
<td>COMPARE AND SIGNAL (short BFP)</td>
<td>RXE C</td>
<td>A</td>
<td>Db Xi</td>
</tr>
<tr>
<td>KIMD</td>
<td>COMPUTE INTERMEDIATE MESSAGE DIGEST</td>
<td>RXE C</td>
<td>MS</td>
<td>A SP</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 5 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLMD</td>
<td>COMPUTE LAST MESSAGE DIGEST</td>
<td>RRE C MS A SP GM I1 ST</td>
<td>R2</td>
<td>7-84</td>
</tr>
<tr>
<td>KM</td>
<td>CIPHER MESSAGE</td>
<td>RRE C MS A SP GM I1 ST</td>
<td>R2</td>
<td>7-35</td>
</tr>
<tr>
<td>KMAC</td>
<td>COMPUTE MESSAGE AUTHENTICATION CODE</td>
<td>RRE C MS A SP GM I1 ST</td>
<td>R2</td>
<td>7-91</td>
</tr>
<tr>
<td>KMC</td>
<td>CIPHER MESSAGE WITH CHAINING</td>
<td>RRE C MS A SP GM I1 ST</td>
<td>R1</td>
<td>7-35</td>
</tr>
<tr>
<td>KXBR</td>
<td>COMPARE AND SIGNAL (extended BFP)</td>
<td>RRE C SP Db Xi</td>
<td>B93F</td>
<td>19-24</td>
</tr>
<tr>
<td>L</td>
<td>LOAD (32)</td>
<td>RX A</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LA</td>
<td>LOAD ADDRESS</td>
<td>RX</td>
<td>50</td>
<td>7-124</td>
</tr>
<tr>
<td>LAE</td>
<td>LOAD ADDRESS EXTENDED</td>
<td>RX</td>
<td>U1 BP 9A</td>
<td>7-124</td>
</tr>
<tr>
<td>LAM</td>
<td>LOAD ACCESS MULTIPLE</td>
<td>RS A SP</td>
<td>UB</td>
<td>7-123</td>
</tr>
<tr>
<td>LAMY</td>
<td>LOAD ACCESS MULTIPLE</td>
<td>RSY LD A SP</td>
<td>UB</td>
<td>7-123</td>
</tr>
<tr>
<td>LARL</td>
<td>LOAD ADDRESS RELATIVE LONG</td>
<td>RIL N3 P A1 SP SO</td>
<td>B1</td>
<td>7-125</td>
</tr>
<tr>
<td>LASP</td>
<td>LOAD ADDRESS SPACE PARAMETERS</td>
<td>SSE C P A1 SP SO</td>
<td>B1</td>
<td>10-36</td>
</tr>
<tr>
<td>LAY</td>
<td>LOAD BYTE (32)</td>
<td>RXY LD A Da</td>
<td>B348</td>
<td>19-34</td>
</tr>
<tr>
<td>LCDR</td>
<td>LOAD COMPLEMENT (long BFP)</td>
<td>RRE C Da</td>
<td>B2</td>
<td>18-14</td>
</tr>
<tr>
<td>LCCEBR</td>
<td>LOAD COMPLEMENT (long HFP)</td>
<td>RRE C Da</td>
<td>B303</td>
<td>19-34</td>
</tr>
<tr>
<td>LCBR</td>
<td>LOAD COMPLEMENT (short BFP)</td>
<td>RRE C Da</td>
<td>B303</td>
<td>19-34</td>
</tr>
<tr>
<td>LCBR</td>
<td>LOAD COMPLEMENT (short HFP)</td>
<td>RRE C Da</td>
<td>B913</td>
<td>7-126</td>
</tr>
<tr>
<td>LCGFR</td>
<td>LOAD COMPLEMENT (64&lt;32)</td>
<td>RRE C N Da</td>
<td>B903</td>
<td>7-126</td>
</tr>
<tr>
<td>LCFR</td>
<td>LOAD COMPLEMENT (64)</td>
<td>RRE C N Da</td>
<td>B313</td>
<td>19-34</td>
</tr>
<tr>
<td>LCR</td>
<td>LOAD COMPLEMENT (32)</td>
<td>RRE C N Da</td>
<td>B313</td>
<td>19-34</td>
</tr>
<tr>
<td>LCTL</td>
<td>LOAD CONTROL (32)</td>
<td>RS P A SP</td>
<td>B2</td>
<td>10-45</td>
</tr>
<tr>
<td>LCTLG</td>
<td>LOAD CONTROL (64)</td>
<td>RSY N P A SP</td>
<td>B2</td>
<td>10-45</td>
</tr>
<tr>
<td>LCXBR</td>
<td>LOAD COMPLEMENT (extended BFP)</td>
<td>RRE C Da</td>
<td>B363</td>
<td>19-34</td>
</tr>
<tr>
<td>LCXR</td>
<td>LOAD COMPLEMENT (extended HFP)</td>
<td>RRE C Da</td>
<td>B363</td>
<td>19-34</td>
</tr>
<tr>
<td>LD</td>
<td>LOAD (long)</td>
<td>RX A Da</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LDE</td>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>RXE A Da</td>
<td>B2</td>
<td>18-15</td>
</tr>
<tr>
<td>LDEB</td>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>RXE A Da</td>
<td>B2</td>
<td>18-15</td>
</tr>
<tr>
<td>LDER</td>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>RXE A Da</td>
<td>B2</td>
<td>18-15</td>
</tr>
<tr>
<td>LDR</td>
<td>LOAD (long)</td>
<td>RR A Da</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LDXBR</td>
<td>LOAD ROUNDED (extended to long BFP)</td>
<td>RRE SP Db Xi Xu Xu Xx Da</td>
<td>B345</td>
<td>19-38</td>
</tr>
<tr>
<td>LDXR</td>
<td>LOAD ROUNDED (extended to long HFP)</td>
<td>RRE SP Db Xi Xu Xu Xx Da</td>
<td>B345</td>
<td>19-38</td>
</tr>
<tr>
<td>LDY</td>
<td>LOAD (long)</td>
<td>RX A Da</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LE</td>
<td>LOAD (short)</td>
<td>RX A Da</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LEDBR</td>
<td>LOAD ROUNDED (long to short BFP)</td>
<td>RRE Da Db Xi Xu Xu Xx</td>
<td>B344</td>
<td>19-38</td>
</tr>
<tr>
<td>LEDR</td>
<td>LOAD ROUNDED (long to short HFP)</td>
<td>RRE Da Db Xi Xu Xu Xx</td>
<td>B344</td>
<td>19-38</td>
</tr>
<tr>
<td>LER</td>
<td>LOAD (short)</td>
<td>RR Da</td>
<td>B2</td>
<td>9-10</td>
</tr>
<tr>
<td>LEXBR</td>
<td>LOAD ROUNDED (extended to short BFP)</td>
<td>RRE SP Da Db Xi Xu Xu Xx</td>
<td>B346</td>
<td>18-17</td>
</tr>
<tr>
<td>LEXR</td>
<td>LOAD ROUNDED (extended to short HFP)</td>
<td>RRE SP Da Db Xi Xu Xu Xx</td>
<td>B346</td>
<td>18-17</td>
</tr>
<tr>
<td>LEY</td>
<td>LOAD (short)</td>
<td>RXY LD A Da</td>
<td>B2</td>
<td>9-11</td>
</tr>
<tr>
<td>LFPC</td>
<td>LOAD FPC</td>
<td>RXY N A Da</td>
<td>B2</td>
<td>19-36</td>
</tr>
<tr>
<td>LG</td>
<td>LOAD (64)</td>
<td>RXY N A Da</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LGB</td>
<td>LOAD BYTE (64)</td>
<td>RXY LD A Da</td>
<td>B2</td>
<td>7-126</td>
</tr>
<tr>
<td>LGF</td>
<td>LOAD (64&lt;32)</td>
<td>RXY N A Da</td>
<td>B2</td>
<td>7-123</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 6 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGFR</td>
<td>LOAD (64&lt;32)</td>
<td>RRE N</td>
<td>B914</td>
<td>7-123</td>
</tr>
<tr>
<td>LGH</td>
<td>LOAD HALFWORD (64)</td>
<td>RXY N</td>
<td>E315</td>
<td>7-127</td>
</tr>
<tr>
<td>LGHI</td>
<td>LOAD HALFWORD IMMEDIATE (64)</td>
<td>RI N</td>
<td>A79</td>
<td>7-127</td>
</tr>
<tr>
<td>LGR</td>
<td>LOAD (64)</td>
<td>RRE N</td>
<td>B904</td>
<td>7-123</td>
</tr>
<tr>
<td>LH</td>
<td>LOAD HALFWORD (32)</td>
<td>RX</td>
<td>48</td>
<td>7-127</td>
</tr>
<tr>
<td>LHI</td>
<td>LOAD HALFWORD IMMEDIATE (32)</td>
<td>RI</td>
<td>A78</td>
<td>7-127</td>
</tr>
<tr>
<td>LHY</td>
<td>LOAD HALFWORD (32)</td>
<td>RXY LD</td>
<td>E378</td>
<td>7-128</td>
</tr>
<tr>
<td>LLGC</td>
<td>LOAD LOGICAL CHARACTER</td>
<td>RXY N</td>
<td>E390</td>
<td>7-128</td>
</tr>
<tr>
<td>LLGF</td>
<td>LOAD LOGICAL (64&lt;32)</td>
<td>RXY N</td>
<td>E316</td>
<td>7-128</td>
</tr>
<tr>
<td>LLGFR</td>
<td>LOAD LOGICAL (64&lt;32)</td>
<td>RRE N</td>
<td>B916</td>
<td>7-128</td>
</tr>
<tr>
<td>LLGH</td>
<td>LOAD LOGICAL HALFWORD</td>
<td>RXY N</td>
<td>B2</td>
<td>7-128</td>
</tr>
<tr>
<td>LLGT</td>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>RXY N</td>
<td>E391</td>
<td>7-128</td>
</tr>
<tr>
<td>LLGTR</td>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>RRE N</td>
<td>B2</td>
<td>7-129</td>
</tr>
<tr>
<td>LLIH</td>
<td>LOAD LOGICAL IMMEDIATE (high high)</td>
<td>RI N</td>
<td>B2</td>
<td>7-128</td>
</tr>
<tr>
<td>LLIHL</td>
<td>LOAD LOGICAL IMMEDIATE (high low)</td>
<td>RI N</td>
<td>B2</td>
<td>7-128</td>
</tr>
<tr>
<td>LLIHH</td>
<td>LOAD LOGICAL IMMEDIATE (low high)</td>
<td>RI N</td>
<td>B2</td>
<td>7-128</td>
</tr>
<tr>
<td>LLIHL</td>
<td>LOAD LOGICAL IMMEDIATE (low low)</td>
<td>RI N</td>
<td>B2</td>
<td>7-128</td>
</tr>
<tr>
<td>LM</td>
<td>LOAD MULTIPLE (32)</td>
<td>RSY N</td>
<td>B2</td>
<td>7-129</td>
</tr>
<tr>
<td>LMD</td>
<td>LOAD MULTIPLE DISJOINT</td>
<td>RSY LD</td>
<td>EF</td>
<td>7-130</td>
</tr>
<tr>
<td>LMG</td>
<td>LOAD MULTIPLE (64)</td>
<td>RSY N</td>
<td>B2</td>
<td>7-129</td>
</tr>
<tr>
<td>LMH</td>
<td>LOAD MULTIPLE HIGH</td>
<td>RSY N</td>
<td>EB96</td>
<td>7-130</td>
</tr>
<tr>
<td>LMY</td>
<td>LOAD MULTIPLE (32)</td>
<td>RSY LD</td>
<td>EB98</td>
<td>7-129</td>
</tr>
<tr>
<td>LNDBR</td>
<td>LOAD NEGATIVE (long BFP)</td>
<td>RRE C</td>
<td>B311</td>
<td>19-37</td>
</tr>
<tr>
<td>LDDR</td>
<td>LOAD NEGATIVE (long HFP)</td>
<td>RR C</td>
<td>21</td>
<td>18-16</td>
</tr>
<tr>
<td>LNEBR</td>
<td>LOAD NEGATIVE (short BFP)</td>
<td>RRE C</td>
<td>B301</td>
<td>19-37</td>
</tr>
<tr>
<td>LNER</td>
<td>LOAD NEGATIVE (short HFP)</td>
<td>RR C</td>
<td>31</td>
<td>18-16</td>
</tr>
<tr>
<td>LNGFR</td>
<td>LOAD NEGATIVE (64&lt;32)</td>
<td>RRE C</td>
<td>B911</td>
<td>7-131</td>
</tr>
<tr>
<td>LGR</td>
<td>LOAD NEGATIVE (64)</td>
<td>RRE C</td>
<td>B901</td>
<td>7-130</td>
</tr>
<tr>
<td>LNR</td>
<td>LOAD NEGATIVE (32)</td>
<td>RR C</td>
<td>11</td>
<td>7-130</td>
</tr>
<tr>
<td>LNXBR</td>
<td>LOAD NEGATIVE (extended BFP)</td>
<td>RRE C</td>
<td>B341</td>
<td>19-37</td>
</tr>
<tr>
<td>LNXR</td>
<td>LOAD NEGATIVE (extended HFP)</td>
<td>RRE C</td>
<td>B361</td>
<td>18-16</td>
</tr>
<tr>
<td>LPDBR</td>
<td>LOAD POSITIVE (long BFP)</td>
<td>RRE C</td>
<td>B310</td>
<td>19-38</td>
</tr>
<tr>
<td>LPDR</td>
<td>LOAD POSITIVE (long HFP)</td>
<td>RR C</td>
<td>20</td>
<td>18-16</td>
</tr>
<tr>
<td>LPEBR</td>
<td>LOAD POSITIVE (short BFP)</td>
<td>RR C</td>
<td>B300</td>
<td>19-38</td>
</tr>
<tr>
<td>LPER</td>
<td>LOAD POSITIVE (short HFP)</td>
<td>RR C</td>
<td>30</td>
<td>18-16</td>
</tr>
<tr>
<td>LPGFR</td>
<td>LOAD POSITIVE (64&lt;32)</td>
<td>RRE C</td>
<td>B910</td>
<td>7-131</td>
</tr>
<tr>
<td>LPGR</td>
<td>LOAD POSITIVE (64)</td>
<td>RRE C</td>
<td>B900</td>
<td>7-131</td>
</tr>
<tr>
<td>LPQ</td>
<td>LOAD PAIR FROM QUADWORD</td>
<td>RXY N</td>
<td>E38F</td>
<td>7-131</td>
</tr>
<tr>
<td>LPR</td>
<td>LOAD POSITIVE (32)</td>
<td>RR C</td>
<td>10</td>
<td>7-131</td>
</tr>
<tr>
<td>LPSW</td>
<td>LOAD PSW</td>
<td>S L</td>
<td>B2</td>
<td>10-45</td>
</tr>
<tr>
<td>LPSWE</td>
<td>LOAD PSW EXTENDED</td>
<td>S L N</td>
<td>B282</td>
<td>10-46</td>
</tr>
<tr>
<td>LPXBR</td>
<td>LOAD POSITIVE (extended BFP)</td>
<td>RRE C</td>
<td>B340</td>
<td>19-38</td>
</tr>
<tr>
<td>LPXR</td>
<td>LOAD POSITIVE (extended HFP)</td>
<td>RRE C</td>
<td>B360</td>
<td>18-16</td>
</tr>
<tr>
<td>LR</td>
<td>LOAD (32)</td>
<td>RR</td>
<td>18</td>
<td>7-123</td>
</tr>
<tr>
<td>LRA</td>
<td>LOAD REAL ADDRESS (32)</td>
<td>RX C</td>
<td>BP</td>
<td>10-47</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 7 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRAG</td>
<td>LOAD REAL ADDRESS (64)</td>
<td>RXY C N</td>
<td>E303</td>
<td>10-47</td>
</tr>
<tr>
<td>LRAY</td>
<td>LOAD REAL ADDRESS (32)</td>
<td>RXY C LD</td>
<td>E313</td>
<td>10-47</td>
</tr>
<tr>
<td>LDRD</td>
<td>LOAD ROUNDED (extended to long HFP)</td>
<td>RR SP</td>
<td>25</td>
<td>18-17</td>
</tr>
<tr>
<td>LRER</td>
<td>LOAD ROUNDED (long to short HFP)</td>
<td>RXY Y N3 A</td>
<td>35</td>
<td>18-17</td>
</tr>
<tr>
<td>LRV</td>
<td>LOAD REVERSED (32)</td>
<td>RXY C N</td>
<td>B2</td>
<td>7-132</td>
</tr>
<tr>
<td>LRVG</td>
<td>LOAD REVERSED (64)</td>
<td>RXY N</td>
<td>E30F</td>
<td>7-132</td>
</tr>
<tr>
<td>LRVGR</td>
<td>LOAD REVERSED (64)</td>
<td>RRE N</td>
<td>B90F</td>
<td>7-132</td>
</tr>
<tr>
<td>LRVH</td>
<td>LOAD REVERSED (16)</td>
<td>RXY N3 A</td>
<td>E31F</td>
<td>7-132</td>
</tr>
<tr>
<td>LRVR</td>
<td>LOAD REVERSED (32)</td>
<td>RRE N</td>
<td>B91F</td>
<td>7-132</td>
</tr>
<tr>
<td>LTDBR</td>
<td>LOAD AND TEST (long BFP)</td>
<td>RRE C</td>
<td>B312</td>
<td>19-34</td>
</tr>
<tr>
<td>LTDR</td>
<td>LOAD AND TEST (long HFP)</td>
<td>RR C</td>
<td>22</td>
<td>18-14</td>
</tr>
<tr>
<td>LTEBR</td>
<td>LOAD AND TEST (short BFP)</td>
<td>RRE C</td>
<td>B302</td>
<td>19-34</td>
</tr>
<tr>
<td>LTER</td>
<td>LOAD AND TEST (short HFP)</td>
<td>RRE N</td>
<td>32</td>
<td>19-34</td>
</tr>
<tr>
<td>LTGFR</td>
<td>LOAD AND TEST (64&lt;32)</td>
<td>RRE C</td>
<td>B912</td>
<td>7-126</td>
</tr>
<tr>
<td>LTGR</td>
<td>LOAD AND TEST (64)</td>
<td>RRE C N</td>
<td>B902</td>
<td>7-126</td>
</tr>
<tr>
<td>LTR</td>
<td>LOAD AND TEST (32)</td>
<td>RRE C</td>
<td>12</td>
<td>7-126</td>
</tr>
<tr>
<td>LTXBR</td>
<td>LOAD AND TEST (extended BFP)</td>
<td>RRE C</td>
<td>B342</td>
<td>19-34</td>
</tr>
<tr>
<td>LTXR</td>
<td>LOAD AND TEST (extended HFP)</td>
<td>RRE C</td>
<td>B362</td>
<td>19-34</td>
</tr>
<tr>
<td>LURA</td>
<td>LOAD USING REAL ADDRESS (32)</td>
<td>RRE</td>
<td>10-52</td>
<td></td>
</tr>
<tr>
<td>LURAG</td>
<td>LOAD USING REAL ADDRESS (64)</td>
<td>RRE N</td>
<td>B905</td>
<td>10-52</td>
</tr>
<tr>
<td>LXD</td>
<td>LOAD LENGTHENED (long to ext. HFP)</td>
<td>RXE A SP</td>
<td>B2</td>
<td>18-15</td>
</tr>
<tr>
<td>LXD B</td>
<td>LOAD LENGTHENED (long to ext. BFP)</td>
<td>RXE A SP</td>
<td>B305</td>
<td>19-37</td>
</tr>
<tr>
<td>LXDR</td>
<td>LOAD LENGTHENED (long to ext. BFP)</td>
<td>RXE A SP</td>
<td>B325</td>
<td>18-15</td>
</tr>
<tr>
<td>LXR</td>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>RXE A SP</td>
<td>B358</td>
<td>9-10</td>
</tr>
<tr>
<td>LXE</td>
<td>LOAD (extended)</td>
<td>RXY LD A</td>
<td>B375</td>
<td>9-11</td>
</tr>
<tr>
<td>LY</td>
<td>LOAD (32)</td>
<td>RXY LD A</td>
<td>B374</td>
<td>9-11</td>
</tr>
<tr>
<td>LZDR</td>
<td>LOAD ZERO (long)</td>
<td>RRE</td>
<td>B376</td>
<td>9-11</td>
</tr>
<tr>
<td>LZER</td>
<td>LOAD ZERO (short)</td>
<td>RRE</td>
<td>B2</td>
<td>7-123</td>
</tr>
<tr>
<td>LZXR</td>
<td>LOAD ZERO (extended)</td>
<td>RRE</td>
<td>B3C</td>
<td>18-19</td>
</tr>
<tr>
<td>M MAD</td>
<td>MULTIPLY (64&lt;32)</td>
<td>RX A SP</td>
<td>B2</td>
<td>18-19</td>
</tr>
<tr>
<td>MAD</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>RXF HM A Da EU EO</td>
<td>B2</td>
<td>18-19</td>
</tr>
<tr>
<td>MAD BR</td>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>RXF HM A Da EU EO</td>
<td>B2</td>
<td>18-19</td>
</tr>
<tr>
<td>MADD</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>RXF HM A Da EU EO</td>
<td>B2</td>
<td>18-19</td>
</tr>
<tr>
<td>MAE</td>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>RXF HM A</td>
<td>B2</td>
<td>18-19</td>
</tr>
<tr>
<td>MAE BR</td>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>RXF HM A</td>
<td>B2</td>
<td>18-19</td>
</tr>
<tr>
<td>MCB</td>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>RXF HM A</td>
<td>B2</td>
<td>18-19</td>
</tr>
<tr>
<td>MC</td>
<td>MONITOR CALL</td>
<td>RX A</td>
<td>B2</td>
<td>18-18</td>
</tr>
<tr>
<td>MD</td>
<td>MULTIPLY (long HFP)</td>
<td>RX A</td>
<td>B2</td>
<td>18-18</td>
</tr>
<tr>
<td>MDB</td>
<td>MULTIPLY (long BFP)</td>
<td>RXE A Da EU EO</td>
<td>B2</td>
<td>19-39</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 8 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDBR</td>
<td>MULTIPLY (long BFP)</td>
<td>RRE</td>
<td>Db Xi Xo Xu Xx</td>
<td>B31C</td>
</tr>
<tr>
<td>MDE</td>
<td>MULTIPLY (short to long HFP)</td>
<td>RX</td>
<td>Da EU EO</td>
<td>7C</td>
</tr>
<tr>
<td>MDEB</td>
<td>MULTIPLY (short to long BFP)</td>
<td>RXE</td>
<td>Db Xi</td>
<td>ED0C</td>
</tr>
<tr>
<td>MDEBR</td>
<td>MULTIPLY (short to long BFP)</td>
<td>RRE</td>
<td>Da EU EO</td>
<td>B30C</td>
</tr>
<tr>
<td>MDER</td>
<td>MULTIPLY (short to long HFP)</td>
<td>RR</td>
<td>Da EU EO</td>
<td>3C</td>
</tr>
<tr>
<td>MDR</td>
<td>MULTIPLY (long HFP)</td>
<td>RR</td>
<td>Da EU EO</td>
<td>2C</td>
</tr>
<tr>
<td>ME</td>
<td>MULTIPLY (short to long HFP)</td>
<td>RX</td>
<td>Da EU EO</td>
<td>B3</td>
</tr>
<tr>
<td>MEE</td>
<td>MULTIPLY (short HFP)</td>
<td>RXE</td>
<td>Db Xi Xo Xu Xx</td>
<td>ED37</td>
</tr>
<tr>
<td>MEEB</td>
<td>MULTIPLY (short BFP)</td>
<td>RXE</td>
<td>Da EU EO</td>
<td>ED17</td>
</tr>
<tr>
<td>MEEBR</td>
<td>MULTIPLY (short BFP)</td>
<td>RRE</td>
<td>Da EU EO</td>
<td>B317</td>
</tr>
<tr>
<td>MEEER</td>
<td>MULTIPLY (short HFP)</td>
<td>RRE</td>
<td>Da EU EO</td>
<td>B337</td>
</tr>
<tr>
<td>MER</td>
<td>MULTIPLY (short to long HFP)</td>
<td>RR</td>
<td>Da EU EO</td>
<td>3C</td>
</tr>
<tr>
<td>MGHI</td>
<td>MULTIPLY HALFWORD IMMEDIATE (64)</td>
<td>RI N</td>
<td></td>
<td>A7D</td>
</tr>
<tr>
<td>MH</td>
<td>MULTIPLY HALFWORD (32)</td>
<td>RX</td>
<td>Da EU EO</td>
<td>4C</td>
</tr>
<tr>
<td>MHI</td>
<td>MULTIPLY HALFWORD IMMEDIATE (32)</td>
<td>RI</td>
<td>Da EU EO</td>
<td>7C</td>
</tr>
<tr>
<td>ML</td>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>RXY N3</td>
<td>Da EU EO</td>
<td>B396</td>
</tr>
<tr>
<td>MLG</td>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>RXY N</td>
<td></td>
<td>B386</td>
</tr>
<tr>
<td>MLGR</td>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B396</td>
</tr>
<tr>
<td>MLR</td>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B396</td>
</tr>
<tr>
<td>MP</td>
<td>MULTIPLY DECIMAL</td>
<td>SS</td>
<td>Dd</td>
<td>FC</td>
</tr>
<tr>
<td>MR</td>
<td>MULTIPLY (64&lt;32)</td>
<td>RR</td>
<td>Da EU EO</td>
<td>1C</td>
</tr>
<tr>
<td>MS</td>
<td>MULTIPLY SINGLE (32)</td>
<td>RX</td>
<td>Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>MSCH</td>
<td>MODIFY SUBCHANNEL</td>
<td>S C</td>
<td>Da EU EO</td>
<td>B322</td>
</tr>
<tr>
<td>MSD</td>
<td>MULTIPLY AND SUBTRACT (long HFP)</td>
<td>RRF HM</td>
<td>Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>MSDBR</td>
<td>MULTIPLY AND SUBTRACT (long BFP)</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B31F</td>
</tr>
<tr>
<td>MSDBR</td>
<td>MULTIPLY AND SUBTRACT (long BFP)</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B31F</td>
</tr>
<tr>
<td>MSER</td>
<td>MULTIPLY AND SUBTRACT (short HFP)</td>
<td>RRF</td>
<td>Da EU EO</td>
<td>B32F</td>
</tr>
<tr>
<td>MSG</td>
<td>MULTIPLY SINGLE (64)</td>
<td>RXY N</td>
<td>Da EU EO</td>
<td>B39C</td>
</tr>
<tr>
<td>MSGF</td>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>RXY N</td>
<td>Da EU EO</td>
<td>B31C</td>
</tr>
<tr>
<td>MSGFR</td>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B91C</td>
</tr>
<tr>
<td>MSGR</td>
<td>MULTIPLY SINGLE (64)</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B90C</td>
</tr>
<tr>
<td>MSR</td>
<td>MULTIPLY SINGLE (32)</td>
<td>RRF</td>
<td>Db Xi Xo Xu Xx</td>
<td>B252</td>
</tr>
<tr>
<td>MSTA</td>
<td>MODIFY STACKED STATE</td>
<td>RRE</td>
<td>Da EU EO</td>
<td>B247</td>
</tr>
<tr>
<td>MSY</td>
<td>MULTIPLY SINGLE (32)</td>
<td>RXY LD</td>
<td>Da EU EO</td>
<td>E351</td>
</tr>
<tr>
<td>MVC</td>
<td>MOVE (character)</td>
<td>SS</td>
<td>Da EU EO</td>
<td>ST</td>
</tr>
<tr>
<td>MVCDE</td>
<td>MOVE WITH DESTINATION KEY</td>
<td>SSE Q</td>
<td>Da EU EO</td>
<td>ST</td>
</tr>
<tr>
<td>MVCK</td>
<td>MOVE INVERSE</td>
<td>SS</td>
<td>Da EU EO</td>
<td>B247</td>
</tr>
<tr>
<td>MVCL</td>
<td>MOVE WITH KEY</td>
<td>SS</td>
<td>Da EU EO</td>
<td>ST</td>
</tr>
<tr>
<td>MVCLE</td>
<td>MOVE LONG</td>
<td>RR</td>
<td>Da EU EO</td>
<td>ST</td>
</tr>
<tr>
<td>MVCLD</td>
<td>MOVE LONG EXTENDED</td>
<td>RS</td>
<td>Da EU EO</td>
<td>ST</td>
</tr>
<tr>
<td>MVCLE</td>
<td>MOVE LONG</td>
<td>RS</td>
<td>Da EU EO</td>
<td>ST</td>
</tr>
<tr>
<td>MVCLUD</td>
<td>MOVE LONG</td>
<td>RS</td>
<td>Da EU EO</td>
<td>ST</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 9 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVCP</td>
<td>MOVE TO PRIMARY</td>
<td>SS C Q A SO $</td>
<td>ST</td>
<td>DA</td>
</tr>
<tr>
<td>MVCS</td>
<td>MOVE TO SECONDARY</td>
<td>SS C Q A SO $</td>
<td>ST</td>
<td>DB</td>
</tr>
<tr>
<td>MVSCK</td>
<td>MOVE WITH SOURCE KEY</td>
<td>SSE Q A GM</td>
<td>ST</td>
<td>E50E</td>
</tr>
<tr>
<td>MVI</td>
<td>MOVE (immediate)</td>
<td>SI A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MVIY</td>
<td>MOVE (immediate)</td>
<td>SIY LD A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MVN</td>
<td>MOVE NUMERICS</td>
<td>SS A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MVO</td>
<td>MOVE WITH OFFSET</td>
<td>SS A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MVPG</td>
<td>MOVE PAGE</td>
<td>RRE C Q A SP G0</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MVST</td>
<td>MOVE STRING</td>
<td>RRE C A SP G0</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MVZ</td>
<td>MOVE ZONES</td>
<td>SS A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>MXBR</td>
<td>MULTIPLY (extended BFP)</td>
<td>RRE SP Da Xi Xo Xu Xx</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>MXD</td>
<td>MULTIPLY (long to extended HFP)</td>
<td>RX A SP Da EU EO</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>MXDB</td>
<td>MULTIPLY (long to extended BFP)</td>
<td>RXE A SP Db Xi</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>MXDBR</td>
<td>MULTIPLY (long to extended BFP)</td>
<td>RRE SP Db Xi</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>MXDR</td>
<td>MULTIPLY (long to extended HFP)</td>
<td>RR SP Da EU EO</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>MXR</td>
<td>MULTIPLY (extended HFP)</td>
<td>RR A Da EU EO</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>N</td>
<td>AND (32)</td>
<td>RR C A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>NC</td>
<td>AND (character)</td>
<td>SS C A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>NG</td>
<td>AND (64)</td>
<td>RXY C N A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>NGR</td>
<td>AND (64)</td>
<td>RRE C N</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>NI</td>
<td>AND (immediate)</td>
<td>SI C A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>NIH</td>
<td>AND IMMEDIATE (high high)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>NII</td>
<td>AND IMMEDIATE (high low)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>NIIH</td>
<td>AND IMMEDIATE (low high)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>NIL</td>
<td>AND IMMEDIATE (low low)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>NIIH</td>
<td>AND IMMEDIATE (low low)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>NIIY</td>
<td>AND (immediate)</td>
<td>SIY C LD A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>NR</td>
<td>AND (32)</td>
<td>RR C A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>NY</td>
<td>AND (32)</td>
<td>RXY C LD A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>O</td>
<td>OR (32)</td>
<td>RX C A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>OC</td>
<td>OR (character)</td>
<td>SS C A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>OG</td>
<td>OR (64)</td>
<td>RXY C N A</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>OGR</td>
<td>OR (64)</td>
<td>RRE C N</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>OI</td>
<td>OR (immediate)</td>
<td>SI C A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>OIHH</td>
<td>OR IMMEDIATE (high high)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>OIHL</td>
<td>OR IMMEDIATE (high low)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>OILH</td>
<td>OR IMMEDIATE (low high)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>OIIL</td>
<td>OR IMMEDIATE (low low)</td>
<td>RI C N</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>OIY</td>
<td>OR (immediate)</td>
<td>SIY C LD A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>OR</td>
<td>OR (32)</td>
<td>RR C A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>OY</td>
<td>OR (32)</td>
<td>RXY C LD A</td>
<td>ST</td>
<td>B1</td>
</tr>
<tr>
<td>PACK</td>
<td>PACK</td>
<td>SS A</td>
<td>ST</td>
<td>B1 B2</td>
</tr>
<tr>
<td>PALB</td>
<td>PURGE ALB</td>
<td>RRE P</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>PC</td>
<td>PROGRAM CALL</td>
<td>S Q A $ Z1 T G M $ B ST</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>PGIN</td>
<td>PAGE IN</td>
<td>RRE C ES P A $</td>
<td>ST</td>
<td>B2</td>
</tr>
<tr>
<td>PGOUT</td>
<td>PAGE OUT</td>
<td>RRE C ES P A $</td>
<td>ST</td>
<td>B2</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 10 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKA</td>
<td>PACK ASCII</td>
<td>SS E2 A SP</td>
<td>ST</td>
<td>B</td>
</tr>
<tr>
<td>PKU</td>
<td>PACK UNICODE</td>
<td>SS E2 A SP</td>
<td>ST</td>
<td>B</td>
</tr>
<tr>
<td>PLO</td>
<td>PERFORM LOCKED OPERATION</td>
<td>SS C A SP $ GM</td>
<td>ST</td>
<td>B</td>
</tr>
<tr>
<td>PR</td>
<td>PROGRAM RETURN</td>
<td>E L Q A SP Z4 T c2</td>
<td>B ST</td>
<td>B</td>
</tr>
<tr>
<td>PT</td>
<td>PROGRAM TRANSFER</td>
<td>RRE Q A SP Z2 T c</td>
<td>B</td>
<td>ST</td>
</tr>
<tr>
<td>PTI</td>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>RRE RA Q A SP WE T</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>PTLB</td>
<td>PURGE TLB</td>
<td>S C P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCHP</td>
<td>RESET CHANNEL PATH</td>
<td>S N P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RLL</td>
<td>ROTATE LEFT SINGLE LOGICAL (32)</td>
<td>RSY N3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RLLG</td>
<td>ROTATE LEFT SINGLE LOGICAL (64)</td>
<td>RSY N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RP</td>
<td>RESUME PROGRAM</td>
<td>S L Q A SP WE T</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>RRBE</td>
<td>RESET REFERENCE BIT EXTENDED</td>
<td>S L Q A SP WE T</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>RSCH</td>
<td>RESUME SUBCHANNEL</td>
<td>S C P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>SUBTRACT (32)</td>
<td>S Q SP IF SW</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>SACF</td>
<td>SET ADDRESS SPACE CONTROL</td>
<td>S</td>
<td>Q SP SW</td>
<td></td>
</tr>
<tr>
<td>SACF</td>
<td>SET ADDRESS SPACE CONTROL FAST</td>
<td>S</td>
<td>Q SP SW</td>
<td></td>
</tr>
<tr>
<td>S anyhow</td>
<td>SUBTRACT (64)</td>
<td>S</td>
<td>A Da EU EO LS</td>
<td></td>
</tr>
<tr>
<td>S anyhow</td>
<td>SUBTRACT (64&lt;32)</td>
<td>S</td>
<td>A Da EU EO LS</td>
<td></td>
</tr>
<tr>
<td>S anyhow</td>
<td>SUBTRACT (64&lt;32)</td>
<td>S</td>
<td>A Da EU EO LS</td>
<td></td>
</tr>
<tr>
<td>SH</td>
<td>SUBTRACT HALFWORD</td>
<td>RXY C LD A IF</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>SIGP</td>
<td>SIGNAL PROCESSOR</td>
<td>RS C P IF</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 11 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLBG</td>
<td>SUBTRACT LOGICAL WITH BORROW (64)</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLBGR</td>
<td>SUBTRACT LOGICAL WITH BORROW (64)</td>
<td>RRE C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLBR</td>
<td>SUBTRACT LOGICAL WITH BORROW (32)</td>
<td>RRE C N3</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLDA</td>
<td>SHIFT LEFT DOUBLE</td>
<td>RS C</td>
<td>SP</td>
<td>IF</td>
</tr>
<tr>
<td>SDL</td>
<td>SHIFT LEFT DOUBLE LOGICAL</td>
<td>RS</td>
<td>SP</td>
<td>B2</td>
</tr>
<tr>
<td>SLG</td>
<td>SUBTRACT LOGICAL (64)</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLGF</td>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLGFR</td>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>RRE C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLL</td>
<td>SHIFT LEFT SINGLE LOGICAL (32)</td>
<td>RS</td>
<td>SP</td>
<td>B2</td>
</tr>
<tr>
<td>SLE</td>
<td>SHIFT LEFT SINGLE LOGICAL (64)</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLEF</td>
<td>SHIFT LEFT SINGLE LOGICAL (64&lt;32)</td>
<td>RXY C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLGR</td>
<td>SHIFT LEFT SINGLE LOGICAL (64)</td>
<td>RRE C N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SLLG</td>
<td>SHIFT LEFT SINGLE LOGICAL (32)</td>
<td>RSY N</td>
<td>A</td>
<td>B2</td>
</tr>
<tr>
<td>SP</td>
<td>SUBTRACT DECIMAL</td>
<td>SS C</td>
<td>A</td>
<td>Dd DF</td>
</tr>
<tr>
<td>SPKA</td>
<td>SET PSW KEY FROM ADDRESS</td>
<td>S</td>
<td>Q</td>
<td>B2</td>
</tr>
<tr>
<td>SPM</td>
<td>SET PROGRAM MASK</td>
<td>RR</td>
<td>P A</td>
<td>$</td>
</tr>
<tr>
<td>SPT</td>
<td>SET CPU TIMER</td>
<td>S</td>
<td>P A</td>
<td>B2</td>
</tr>
<tr>
<td>SPX</td>
<td>SET PREFIX</td>
<td>S</td>
<td>P A</td>
<td>B2</td>
</tr>
<tr>
<td>SQD</td>
<td>SQUARE ROOT (long HFP)</td>
<td>RXE</td>
<td>A Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SQDB</td>
<td>SQUARE ROOT (long BFP)</td>
<td>RXE</td>
<td>A Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>SQDBR</td>
<td>SQUARE ROOT (long BFP)</td>
<td>RRE</td>
<td>Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SQDR</td>
<td>SQUARE ROOT (long HFP)</td>
<td>RRE</td>
<td>Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>SQE</td>
<td>SQUARE ROOT (short HFP)</td>
<td>RXE</td>
<td>A Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SQEB</td>
<td>SQUARE ROOT (short BFP)</td>
<td>RXE</td>
<td>A Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>SQEBR</td>
<td>SQUARE ROOT (short BFP)</td>
<td>RRE</td>
<td>Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SQER</td>
<td>SQUARE ROOT (short HFP)</td>
<td>RRE</td>
<td>Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>SQXBR</td>
<td>SQUARE ROOT (extended BFP)</td>
<td>RRE</td>
<td>Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SQX</td>
<td>SQUARE ROOT (extended HFP)</td>
<td>RRE</td>
<td>Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>SR</td>
<td>SUBTRACT (32)</td>
<td>RR C</td>
<td>IF</td>
<td>B2</td>
</tr>
<tr>
<td>SRAG</td>
<td>SHIFT RIGHT SINGLE (32)</td>
<td>RS C</td>
<td>SP</td>
<td>B2</td>
</tr>
<tr>
<td>SRD</td>
<td>SHIFT RIGHT DOUBLE</td>
<td>RS C</td>
<td>SP</td>
<td>B2</td>
</tr>
<tr>
<td>SRDL</td>
<td>SHIFT RIGHT DOUBLE LOGICAL</td>
<td>RS C</td>
<td>SP</td>
<td>B2</td>
</tr>
<tr>
<td>SRL</td>
<td>SHIFT RIGHT SINGLE LOGICAL (32)</td>
<td>RS</td>
<td>SP</td>
<td>B2</td>
</tr>
<tr>
<td>SRLG</td>
<td>SHIFT RIGHT SINGLE LOGICAL (64)</td>
<td>RSY N</td>
<td>B2</td>
<td>7-180</td>
</tr>
<tr>
<td>SRNM</td>
<td>SET Rounding MODE</td>
<td>S</td>
<td>Db Dd DF</td>
<td>B2</td>
</tr>
<tr>
<td>SRP</td>
<td>SHIFT AND ROUND DECIMAL</td>
<td>SS C</td>
<td>A Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SRST</td>
<td>SEARCH STRING</td>
<td>RRE C</td>
<td>Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SRSTU</td>
<td>SEARCH STRING UNICODE</td>
<td>RRE C</td>
<td>Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>SSAR</td>
<td>SET SECONDARY ASN</td>
<td>RRE</td>
<td>A Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SSSC</td>
<td>SET SECONDARY SUBCHANNEL</td>
<td>RRE</td>
<td>A Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>SSCH</td>
<td>SET STORAGE KEY EXTENDED</td>
<td>RRE</td>
<td>A Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>SSM</td>
<td>SET SYSTEM MASK</td>
<td>S</td>
<td>A Db Xi Xx</td>
<td>B2</td>
</tr>
<tr>
<td>ST</td>
<td>STORE (32)</td>
<td>RX</td>
<td>A Da SQ</td>
<td>B2</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 12 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAM</td>
<td>STORE ACCESS MULTIPLE</td>
<td>RS A SP</td>
<td>ST UB</td>
<td>9B</td>
</tr>
<tr>
<td>STAMY</td>
<td>STORE ACCESS MULTIPLE</td>
<td>RSY LD A SP</td>
<td>ST UB</td>
<td>EB9B</td>
</tr>
<tr>
<td>STAP</td>
<td>STORE CPU ADDRESS</td>
<td>S PA SP</td>
<td>ST Bz</td>
<td>B212</td>
</tr>
<tr>
<td>STC</td>
<td>STORE CHARACTER</td>
<td>RX A</td>
<td>ST Bz</td>
<td>42</td>
</tr>
<tr>
<td>STCK</td>
<td>STORE CLOCK</td>
<td>S C A</td>
<td>ST Bz</td>
<td>B2065</td>
</tr>
<tr>
<td>STCKC</td>
<td>STORE CLOCK COMPARATOR</td>
<td>S P A SP</td>
<td>ST Bz</td>
<td>B207</td>
</tr>
<tr>
<td>STCKE</td>
<td>STORE CLOCK EXTENDED</td>
<td>S C A</td>
<td>ST Bz</td>
<td>B278</td>
</tr>
<tr>
<td>STCM</td>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>RS A</td>
<td>ST Bz</td>
<td>BE</td>
</tr>
<tr>
<td>STCMH</td>
<td>STORE CHARACTERS UNDER MASK (high)</td>
<td>RSY N A</td>
<td>ST Bz</td>
<td>EB2C</td>
</tr>
<tr>
<td>STCMY</td>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>RSY LD A</td>
<td>ST Bz</td>
<td>EB2D</td>
</tr>
<tr>
<td>STCPS</td>
<td>STORE CHANNEL PATH STATUS</td>
<td>S P A SP</td>
<td>ST Bz</td>
<td>B23A</td>
</tr>
<tr>
<td>STCRW</td>
<td>STORE CHANNEL REPORT WORD</td>
<td>S C P A SP</td>
<td>ST Bz</td>
<td>B239</td>
</tr>
<tr>
<td>STCTG</td>
<td>STORE CONTROL (64)</td>
<td>RSY N P A SP</td>
<td>ST Bz</td>
<td>EB25</td>
</tr>
<tr>
<td>STCTL</td>
<td>STORE CONTROL (32)</td>
<td>RS P A SP</td>
<td>ST Bz</td>
<td>B6</td>
</tr>
<tr>
<td>STCY</td>
<td>STORE CHARACTER</td>
<td>RXY LD A</td>
<td>ST E372</td>
<td>7-181</td>
</tr>
<tr>
<td>STD</td>
<td>STORE (long)</td>
<td>RX A</td>
<td>Da</td>
<td>ST B2/zerodot2</td>
</tr>
<tr>
<td>STDY</td>
<td>STORE (long)</td>
<td>RXY LD A</td>
<td>Da</td>
<td>ST E324</td>
</tr>
<tr>
<td>STE</td>
<td>STORE (short)</td>
<td>RX A</td>
<td>Da</td>
<td>ST B2/zerodot7</td>
</tr>
<tr>
<td>STEY</td>
<td>STORE (short)</td>
<td>RXY LD A</td>
<td>Da</td>
<td>ST B2/zerodot5</td>
</tr>
<tr>
<td>STFL</td>
<td>STORE FACILITY LIST</td>
<td>S N3 P</td>
<td>ST B2B1</td>
<td>1/zerodot-11/zerodot</td>
</tr>
<tr>
<td>STG</td>
<td>STORE FPC</td>
<td>S A</td>
<td>Da</td>
<td>ST B29C</td>
</tr>
<tr>
<td>STH</td>
<td>STORE HALFWORD</td>
<td>RXY N A</td>
<td>ST E324</td>
<td>7-180</td>
</tr>
<tr>
<td>STHY</td>
<td>STORE HALFWORD</td>
<td>RXY LD A</td>
<td>ST E325</td>
<td>7-185</td>
</tr>
<tr>
<td>STIDP</td>
<td>STORE CPU ID</td>
<td>S P A</td>
<td>ST B2/zerodot2</td>
<td>1/zerodot-1/zerodot2</td>
</tr>
<tr>
<td>STM</td>
<td>STORE MULTIPLE (32)</td>
<td>RS A</td>
<td>ST B2/zerodot9</td>
<td>1/zerodot-1/zerodot9</td>
</tr>
<tr>
<td>STMG</td>
<td>STORE MULTIPLE (64)</td>
<td>RSY N A</td>
<td>ST EB24</td>
<td>7-185</td>
</tr>
<tr>
<td>STMH</td>
<td>STORE MULTIPLE HIGH</td>
<td>RSY N A</td>
<td>ST EB26</td>
<td>7-186</td>
</tr>
<tr>
<td>STMY</td>
<td>STORE MULTIPLE (32)</td>
<td>RSY LD A</td>
<td>ST EB90</td>
<td>7-185</td>
</tr>
<tr>
<td>STNSM</td>
<td>STORE THEN AND SYSTEM MASK</td>
<td>SI P A</td>
<td>ST E370</td>
<td>7-185</td>
</tr>
<tr>
<td>STOSM</td>
<td>STORE THEN OR SYSTEM MASK</td>
<td>SI P A SP</td>
<td>ST B2/zerodot7</td>
<td>1/zerodot-1/zerodot7</td>
</tr>
<tr>
<td>STPQ</td>
<td>STORE PAIR TO QUADWORD</td>
<td>RXY N A</td>
<td>ST B2/zerodot6</td>
<td>1/zerodot-1/zerodot6</td>
</tr>
<tr>
<td>STPT</td>
<td>STORE CPU TIMER</td>
<td>S P A</td>
<td>ST B2/zerodot5</td>
<td>1/zerodot-1/zerodot5</td>
</tr>
<tr>
<td>STPX</td>
<td>STORE PREFIX</td>
<td>S P A</td>
<td>ST E372</td>
<td>7-187</td>
</tr>
<tr>
<td>STRAG</td>
<td>STORE REAL ADDRESS</td>
<td>SSE N P A</td>
<td>ST B2/zerodot2</td>
<td>1/zerodot-1/zerodot2</td>
</tr>
<tr>
<td>STRV</td>
<td>STORE REVERSED (32)</td>
<td>RXN3 A</td>
<td>ST B2/zerodot2</td>
<td>1/zerodot-1/zerodot2</td>
</tr>
<tr>
<td>STRVG</td>
<td>STORE REVERSED (64)</td>
<td>RXN3 A</td>
<td>ST B2/zerodot2</td>
<td>1/zerodot-1/zerodot2</td>
</tr>
<tr>
<td>STRVR</td>
<td>STORE REVERSED (16)</td>
<td>RXN3 A</td>
<td>ST B2/zerodot2</td>
<td>1/zerodot-1/zerodot2</td>
</tr>
<tr>
<td>STSCH</td>
<td>STORE SUBCHANNEL</td>
<td>S C P A SP</td>
<td>ST B2/zerodot2</td>
<td>1/zerodot-1/zerodot2</td>
</tr>
<tr>
<td>STSI</td>
<td>STORE SYSTEM INFORMATION</td>
<td>S C P A SP</td>
<td>ST B2/zerodot2</td>
<td>1/zerodot-1/zerodot2</td>
</tr>
<tr>
<td>STURA</td>
<td>STORE USING REAL ADDRESS (32)</td>
<td>RRE P A SP</td>
<td>ST B246</td>
<td>10-124</td>
</tr>
<tr>
<td>STURG</td>
<td>STORE USING REAL ADDRESS (64)</td>
<td>RRE N P A SP</td>
<td>ST B925</td>
<td>10-124</td>
</tr>
<tr>
<td>STY</td>
<td>STORE (32)</td>
<td>RXY LD A</td>
<td>ST E350</td>
<td>7-180</td>
</tr>
<tr>
<td>SU</td>
<td>SUBTRACT UNNORMALIZED (short HFP)</td>
<td>RX C A</td>
<td>Da EO LS</td>
<td>ST B2/zerodot2</td>
</tr>
<tr>
<td>SUR</td>
<td>SUBTRACT UNNORMALIZED (short HFP)</td>
<td>RR C A</td>
<td>Da EO LS</td>
<td>ST B2/zerodot2</td>
</tr>
</tbody>
</table>

*Figure B-2 (Part 13 of 15). Instructions Arranged by Mnemonic*
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVC</td>
<td>SUPERVISOR CALL</td>
<td></td>
<td>OA</td>
<td>7-190</td>
</tr>
<tr>
<td>SW</td>
<td>SUBTRACT UNNORMALIZED (long HFP)</td>
<td></td>
<td>0A</td>
<td>18-22</td>
</tr>
<tr>
<td>SWR</td>
<td>SUBTRACT UNNORMALIZED (long HFP)</td>
<td></td>
<td>2F</td>
<td>18-22</td>
</tr>
<tr>
<td>SXBR</td>
<td>SUBTRACT (extended BFP)</td>
<td></td>
<td>B34B</td>
<td>19-44</td>
</tr>
<tr>
<td>SXR</td>
<td>SUBTRACT NORMALIZED (extended HFP)</td>
<td></td>
<td>37</td>
<td>18-22</td>
</tr>
<tr>
<td>SY</td>
<td>SUBTRACT (32)</td>
<td></td>
<td>B2</td>
<td>7-187</td>
</tr>
<tr>
<td>TAM</td>
<td>TEST ADDRESSING MODE</td>
<td></td>
<td>E35B</td>
<td>7-190</td>
</tr>
<tr>
<td>TAR</td>
<td>TEST ACCESS</td>
<td></td>
<td>010B</td>
<td>10-125</td>
</tr>
<tr>
<td>TB</td>
<td>TEST BLOCK</td>
<td></td>
<td>B24C</td>
<td>10-127</td>
</tr>
<tr>
<td>TBDR</td>
<td>CONVERT HFP TO BFP (long)</td>
<td></td>
<td>B351</td>
<td>9-9</td>
</tr>
<tr>
<td>TBEDR</td>
<td>CONVERT HFP TO BFP (long to short)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCB</td>
<td>TEST DATA CLASS (long BFP)</td>
<td></td>
<td>E11</td>
<td>19-45</td>
</tr>
<tr>
<td>TCEB</td>
<td>TEST DATA CLASS (short BFP)</td>
<td></td>
<td>E10</td>
<td>19-45</td>
</tr>
<tr>
<td>TCXB</td>
<td>TEST DATA CLASS (extended BFP)</td>
<td></td>
<td>E12</td>
<td>19-45</td>
</tr>
<tr>
<td>TDBR</td>
<td>CONVERT HFP TO BFP (short to long)</td>
<td></td>
<td>B358</td>
<td>9-8</td>
</tr>
<tr>
<td>THDR</td>
<td>CONVERT BFP TO HFP (long)</td>
<td></td>
<td>B359</td>
<td>9-8</td>
</tr>
<tr>
<td>TM</td>
<td>TEST UNDER MASK</td>
<td></td>
<td>A71</td>
<td>7-191</td>
</tr>
<tr>
<td>TMH</td>
<td>TEST UNDER MASK (low)</td>
<td></td>
<td>A72</td>
<td>7-191</td>
</tr>
<tr>
<td>TMHL</td>
<td>TEST UNDER MASK (low high)</td>
<td></td>
<td>A73</td>
<td>7-191</td>
</tr>
<tr>
<td>TML</td>
<td>TEST UNDER MASK LOW</td>
<td></td>
<td>A71</td>
<td>7-192</td>
</tr>
<tr>
<td>TMLH</td>
<td>TEST UNDER MASK (low high)</td>
<td></td>
<td>A70</td>
<td>7-192</td>
</tr>
<tr>
<td>TMLL</td>
<td>TEST UNDER MASK (low low)</td>
<td></td>
<td>A72</td>
<td>7-192</td>
</tr>
<tr>
<td>TMY</td>
<td>TEST UNDER MASK</td>
<td></td>
<td>EB51</td>
<td>7-191</td>
</tr>
<tr>
<td>TP</td>
<td>TEST DECIMAL</td>
<td></td>
<td>EBC0</td>
<td>8-13</td>
</tr>
<tr>
<td>TPI</td>
<td>TEST PENDING INTERRUPTION</td>
<td></td>
<td>B236</td>
<td>14-18</td>
</tr>
<tr>
<td>TPROT</td>
<td>TEST PROTECTION</td>
<td></td>
<td>E501</td>
<td>10-129</td>
</tr>
<tr>
<td>TR</td>
<td>TRANSLATE</td>
<td></td>
<td>B2DC</td>
<td>7-192</td>
</tr>
<tr>
<td>TRACE</td>
<td>TRACE (32)</td>
<td></td>
<td>B299</td>
<td>10-132</td>
</tr>
<tr>
<td>TRACG</td>
<td>TRACE (64)</td>
<td></td>
<td>EB6F</td>
<td>10-132</td>
</tr>
<tr>
<td>TRAP2</td>
<td>TRAP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP4</td>
<td>TRAP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRE</td>
<td>TRANSLATE EXTENDED</td>
<td></td>
<td>R2A5</td>
<td>7-195</td>
</tr>
<tr>
<td>TROO</td>
<td>TRANSLATE ONE TO ONE</td>
<td></td>
<td>B993</td>
<td>7-198</td>
</tr>
<tr>
<td>TROT</td>
<td>TRANSLATE ONE TO TWO</td>
<td></td>
<td>B992</td>
<td>7-198</td>
</tr>
<tr>
<td>TRT</td>
<td>TRANSLATE AND TEST</td>
<td></td>
<td>B2B2</td>
<td>7-193</td>
</tr>
<tr>
<td>TRTO</td>
<td>TRANSLATE TWO TO ONE</td>
<td></td>
<td>B991</td>
<td>7-198</td>
</tr>
<tr>
<td>TRTR</td>
<td>TRANSLATE TWO TO TWO</td>
<td></td>
<td>B990</td>
<td>7-198</td>
</tr>
<tr>
<td>TS</td>
<td>TEST AND SET</td>
<td></td>
<td>B93</td>
<td>7-191</td>
</tr>
<tr>
<td>TSCH</td>
<td>TEST SUBCHANNEL</td>
<td></td>
<td>B235</td>
<td>14-20</td>
</tr>
<tr>
<td>UNPK</td>
<td>UNPACK</td>
<td></td>
<td>B2F</td>
<td>7-202</td>
</tr>
<tr>
<td>UNPKA</td>
<td>UNPACK ASCII</td>
<td></td>
<td>B2E</td>
<td>7-203</td>
</tr>
<tr>
<td>UNPKU</td>
<td>UNPACK UNICODE</td>
<td></td>
<td>B2E</td>
<td>7-204</td>
</tr>
<tr>
<td>UPT</td>
<td>UPDATE TREE</td>
<td></td>
<td>0102</td>
<td>7-205</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 14 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Characteristics</th>
<th>Op Code</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>EXCLUSIVE OR (32)</td>
<td>RX C A</td>
<td>B₂ B₂</td>
<td>57 7-118</td>
</tr>
<tr>
<td>XC</td>
<td>EXCLUSIVE OR (character)</td>
<td>SS C A</td>
<td>ST B₁ B₂ D7</td>
<td>7-118</td>
</tr>
<tr>
<td>XG</td>
<td>EXCLUSIVE OR (64)</td>
<td>RXY C N A</td>
<td>B₂ E382</td>
<td>7-118</td>
</tr>
<tr>
<td>XGR</td>
<td>EXCLUSIVE OR (64)</td>
<td>RRE C N</td>
<td>B₁ B₂ B₉82</td>
<td>7-118</td>
</tr>
<tr>
<td>XI</td>
<td>EXCLUSIVE OR (immediate)</td>
<td>SI C A</td>
<td>ST B₁</td>
<td>97 7-118</td>
</tr>
<tr>
<td>XIX</td>
<td>EXCLUSIVE OR (immediate)</td>
<td>SIY C LD A</td>
<td>ST B₁ E₆57</td>
<td>7-118</td>
</tr>
<tr>
<td>XR</td>
<td>EXCLUSIVE OR (32)</td>
<td>RR C P OP Æ GS</td>
<td>B₂ B₉76</td>
<td>14-4 7-118</td>
</tr>
<tr>
<td>XSCH</td>
<td>CANCEL SUBCHANNEL</td>
<td>S C</td>
<td>B₁ B₂ E₃57</td>
<td>7-118</td>
</tr>
<tr>
<td>XY</td>
<td>EXCLUSIVE OR (32)</td>
<td>RXY C LD A</td>
<td>B₂ B₂ F₈</td>
<td>8-13</td>
</tr>
<tr>
<td>ZAP</td>
<td>ZERO AND ADD</td>
<td>SS C A Dd DF</td>
<td>ST B₁ B₂</td>
<td>8-13</td>
</tr>
</tbody>
</table>

Figure B-2 (Part 15 of 15). Instructions Arranged by Mnemonic
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
<td>PROGRAM RETURN</td>
<td>PR</td>
<td>E L</td>
<td>10-76</td>
</tr>
<tr>
<td>0102</td>
<td>UPDATE TREE</td>
<td>UPT</td>
<td>E C</td>
<td>7-205</td>
</tr>
<tr>
<td>0107</td>
<td>SET CLOCK PROGRAMMABLE FIELD</td>
<td>SCKPF</td>
<td>E C</td>
<td>10-97</td>
</tr>
<tr>
<td>010B</td>
<td>TEST ADDRESSING MODE</td>
<td>TAM</td>
<td>E C N3</td>
<td>7-190</td>
</tr>
<tr>
<td>010C</td>
<td>SET ADDRESSING MODE (24)</td>
<td>SAM24</td>
<td>E N3</td>
<td>7-175</td>
</tr>
<tr>
<td>010D</td>
<td>SET ADDRESSING MODE (31)</td>
<td>SAM31</td>
<td>E N3</td>
<td>7-175</td>
</tr>
<tr>
<td>010E</td>
<td>SET ADDRESSING MODE (64)</td>
<td>SAM64</td>
<td>E N</td>
<td>7-175</td>
</tr>
<tr>
<td>01FF</td>
<td>TRAP</td>
<td>TRAP2</td>
<td>E A</td>
<td>10-133</td>
</tr>
<tr>
<td>04</td>
<td>SET PROGRAM MASK</td>
<td>SPM</td>
<td>RR L</td>
<td>7-175</td>
</tr>
<tr>
<td>05</td>
<td>BRANCH AND LINK</td>
<td>BALR</td>
<td></td>
<td>7-22</td>
</tr>
<tr>
<td>06</td>
<td>BRANCH ON COUNT (32)</td>
<td>BCTR</td>
<td>RR</td>
<td>7-26</td>
</tr>
<tr>
<td>07</td>
<td>BRANCH ON CONDITION</td>
<td>BCR</td>
<td>RR</td>
<td>7-25</td>
</tr>
<tr>
<td>0A</td>
<td>SUPERVISOR CALL</td>
<td>SVC</td>
<td>I</td>
<td>7-190</td>
</tr>
<tr>
<td>0B</td>
<td>BRANCH AND SET MODE</td>
<td>BSM</td>
<td>RR</td>
<td>7-24</td>
</tr>
<tr>
<td>0C</td>
<td>BRANCH AND SAVE AND SET MODE</td>
<td>BASM</td>
<td>RR</td>
<td>7-23</td>
</tr>
<tr>
<td>0D</td>
<td>BRANCH AND SAVE</td>
<td>BASR</td>
<td>RR C</td>
<td>7-23</td>
</tr>
<tr>
<td>0E</td>
<td>MOVE LONG</td>
<td>MVCL</td>
<td>RR C</td>
<td>7-23</td>
</tr>
<tr>
<td>0F</td>
<td>COMPARE LOGICAL LONG</td>
<td>CLCL</td>
<td>RR C</td>
<td>7-23</td>
</tr>
<tr>
<td>10</td>
<td>LOAD POSITIVE (32)</td>
<td>LPR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>11</td>
<td>LOAD NEGATIVE (32)</td>
<td>LNR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>12</td>
<td>LOAD AND TEST (32)</td>
<td>LTR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>13</td>
<td>LOAD COMPLEMENT (32)</td>
<td>LCR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>14</td>
<td>AND (32)</td>
<td>NR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>15</td>
<td>COMPARE LOGICAL (32)</td>
<td>CLR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>16</td>
<td>OR (32)</td>
<td>OR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>17</td>
<td>EXCLUSIVE OR (32)</td>
<td>XR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>18</td>
<td>LOAD (32)</td>
<td>LR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>19</td>
<td>COMPARE (32)</td>
<td>CR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>1A</td>
<td>ADD (32)</td>
<td>AR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>1B</td>
<td>SUBTRACT (32)</td>
<td>SR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>1C</td>
<td>MULTIPLY (64&lt;32)</td>
<td>MR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>1D</td>
<td>DIVIDE (32&lt;64)</td>
<td>DR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>1E</td>
<td>ADD LOGICAL (32)</td>
<td>ALR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>1F</td>
<td>SUBTRACT LOGICAL (32)</td>
<td>SLR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>20</td>
<td>LOAD POSITIVE (long HFP)</td>
<td>LPDR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>21</td>
<td>LOAD NEGATIVE (long HFP)</td>
<td>LNDR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>22</td>
<td>LOAD AND TEST (long HFP)</td>
<td>LTRD</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>23</td>
<td>LOAD COMPLEMENT (long HFP)</td>
<td>LCRD</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>24</td>
<td>HALVE (long HFP)</td>
<td>HDR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>25</td>
<td>LOAD ROUNDED (extended to long HFP)</td>
<td>LDXR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>26</td>
<td>LOAD ROUNDED (extended to long HFP)</td>
<td>LRDR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>27</td>
<td>MULTIPLY (extended HFP)</td>
<td>MXR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>28</td>
<td>LOAD (long)</td>
<td>LDR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
<tr>
<td>29</td>
<td>COMPARE (long HFP)</td>
<td>CDR</td>
<td>RR C</td>
<td>7-175</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 1 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A</td>
<td>ADD NORMALIZED (long HFP)</td>
<td>ADR</td>
<td>Da EU EO LS</td>
<td>18-8</td>
</tr>
<tr>
<td>2B</td>
<td>SUBTRACT NORMALIZED (long HFP)</td>
<td>SDR</td>
<td>Da EU EO LS</td>
<td>18-22</td>
</tr>
<tr>
<td>2C</td>
<td>MULTIPLY (long HFP)</td>
<td>MDR</td>
<td>Da EU EO</td>
<td>18-18</td>
</tr>
<tr>
<td>2D</td>
<td>DIVIDE (long HFP)</td>
<td>DDR</td>
<td>Da EU EO FK</td>
<td>18-12</td>
</tr>
<tr>
<td>2E</td>
<td>ADD UNNORMALIZED (long HFP)</td>
<td>AWR</td>
<td>Da EO LS</td>
<td>18-10</td>
</tr>
<tr>
<td>2F</td>
<td>SUBTRACT UNNORMALIZED (long HFP)</td>
<td>SWR</td>
<td>Da EO LS</td>
<td>18-22</td>
</tr>
<tr>
<td>3/zerodot</td>
<td>LOAD POSITIVE (short HFP)</td>
<td>LPER</td>
<td>Da</td>
<td>18-16</td>
</tr>
<tr>
<td>31</td>
<td>LOAD NEGATIVE (short HFP)</td>
<td>LNER</td>
<td>Da</td>
<td>18-16</td>
</tr>
<tr>
<td>32</td>
<td>LOAD AND TEST (short HFP)</td>
<td>LTER</td>
<td>Da</td>
<td>18-14</td>
</tr>
<tr>
<td>33</td>
<td>LOAD COMPLEMENT (short HFP)</td>
<td>LCER</td>
<td>Da</td>
<td>18-14</td>
</tr>
<tr>
<td>34</td>
<td>HALVE (short HFP)</td>
<td>HER</td>
<td>Da EU</td>
<td>18-13</td>
</tr>
<tr>
<td>35</td>
<td>LOAD ROUNDED (long to short HFP)</td>
<td>LDER</td>
<td>Da EO</td>
<td>18-17</td>
</tr>
<tr>
<td>36</td>
<td>ADD NORMALIZED (extended HFP)</td>
<td>AXR</td>
<td>Da EU EO LS</td>
<td>18-8</td>
</tr>
<tr>
<td>37</td>
<td>SUBTRACT NORMALIZED (extended HFP)</td>
<td>SXR</td>
<td>Da EU EO LS</td>
<td>18-22</td>
</tr>
<tr>
<td>38</td>
<td>LOAD (short)</td>
<td>LER</td>
<td>Da</td>
<td>9-10</td>
</tr>
<tr>
<td>39</td>
<td>COMPARE (short HFP)</td>
<td>CER</td>
<td>Da</td>
<td>18-10</td>
</tr>
<tr>
<td>3A</td>
<td>ADD NORMALIZED (short HFP)</td>
<td>AER</td>
<td>Da EU EO LS</td>
<td>18-8</td>
</tr>
<tr>
<td>3B</td>
<td>SUBTRACT NORMALIZED (short HFP)</td>
<td>SER</td>
<td>Da</td>
<td>18-22</td>
</tr>
<tr>
<td>3C</td>
<td>MULTIPLY (short to long HFP)</td>
<td>MDER</td>
<td>Da EU EO</td>
<td>18-18</td>
</tr>
<tr>
<td>3D</td>
<td>DIVIDE (short HFP)</td>
<td>DER</td>
<td>Da EU EO FK</td>
<td>18-12</td>
</tr>
<tr>
<td>3E</td>
<td>ADD UNNORMALIZED (short HFP)</td>
<td>AUR</td>
<td>Da EO LS</td>
<td>18-10</td>
</tr>
<tr>
<td>3F</td>
<td>SUBTRACT UNNORMALIZED (short HFP)</td>
<td>SUR</td>
<td>Da EO LS</td>
<td>18-22</td>
</tr>
<tr>
<td>4/zerodot</td>
<td>STORE HALFWORD</td>
<td>STH</td>
<td>ST</td>
<td>7-185</td>
</tr>
<tr>
<td>41</td>
<td>LOAD ADDRESS</td>
<td>LA</td>
<td>ST</td>
<td>7-124</td>
</tr>
<tr>
<td>42</td>
<td>STORE CHARACTER</td>
<td>STC</td>
<td>ST</td>
<td>7-181</td>
</tr>
<tr>
<td>43</td>
<td>INSERT CHARACTER</td>
<td>IC</td>
<td>ST</td>
<td>7-121</td>
</tr>
<tr>
<td>44</td>
<td>EXECUTE</td>
<td>EX</td>
<td>EX</td>
<td>7-119</td>
</tr>
<tr>
<td>45</td>
<td>BRANCH AND LINK</td>
<td>BAL</td>
<td>ST</td>
<td>7-22</td>
</tr>
<tr>
<td>46</td>
<td>BRANCH ON COUNT (32)</td>
<td>BCT</td>
<td>B</td>
<td>7-26</td>
</tr>
<tr>
<td>47</td>
<td>BRANCH ON CONDITION</td>
<td>BC</td>
<td>B</td>
<td>7-25</td>
</tr>
<tr>
<td>48</td>
<td>LOAD HALFWORD (32)</td>
<td>LH</td>
<td>B</td>
<td>7-127</td>
</tr>
<tr>
<td>49</td>
<td>COMPARE HALFWORD</td>
<td>CH</td>
<td>B</td>
<td>7-55</td>
</tr>
<tr>
<td>4A</td>
<td>ADD HALFWORD</td>
<td>AH</td>
<td>B</td>
<td>7-18</td>
</tr>
<tr>
<td>4B</td>
<td>SUBTRACT HALFWORD</td>
<td>SH</td>
<td>IF</td>
<td>7-188</td>
</tr>
<tr>
<td>4C</td>
<td>MULTIPLY HALFWORD (32)</td>
<td>MH</td>
<td>IF</td>
<td>7-149</td>
</tr>
<tr>
<td>4D</td>
<td>BRANCH AND SAVE</td>
<td>BAS</td>
<td>ST</td>
<td>7-98</td>
</tr>
<tr>
<td>4E</td>
<td>CONVERT TO DECIMAL (32)</td>
<td>CVD</td>
<td>ST</td>
<td>7-97</td>
</tr>
<tr>
<td>4F</td>
<td>CONVERT TO BINARY (32)</td>
<td>CVB</td>
<td>ST</td>
<td>7-180</td>
</tr>
<tr>
<td>5/zerodot</td>
<td>STORE (32)</td>
<td>ST</td>
<td>ST</td>
<td>7-180</td>
</tr>
<tr>
<td>51</td>
<td>LOAD ADDRESS EXTENDED</td>
<td>LAE</td>
<td>ST</td>
<td>7-124</td>
</tr>
<tr>
<td>54</td>
<td>AND (32)</td>
<td>N</td>
<td>ST</td>
<td>7-20</td>
</tr>
<tr>
<td>55</td>
<td>COMPARE LOGICAL (32)</td>
<td>CL</td>
<td>ST</td>
<td>7-56</td>
</tr>
<tr>
<td>56</td>
<td>OR (32)</td>
<td>O</td>
<td>ST</td>
<td>7-151</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 2 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>EXCLUSIVE OR (32)</td>
<td>X</td>
<td>RX C A</td>
<td>Bz 7-118</td>
</tr>
<tr>
<td>58</td>
<td>LOAD (32)</td>
<td>L</td>
<td>RX A</td>
<td>Bz 7-123</td>
</tr>
<tr>
<td>59</td>
<td>COMPARE (32)</td>
<td>C</td>
<td>RX C A</td>
<td>Bz 7-46</td>
</tr>
<tr>
<td>5A</td>
<td>ADD (32)</td>
<td>A</td>
<td>RX C A IF</td>
<td>Bz 7-18</td>
</tr>
<tr>
<td>5B</td>
<td>SUBTRACT (32)</td>
<td>S</td>
<td>RX C A IF</td>
<td>Bz 7-187</td>
</tr>
<tr>
<td>5C</td>
<td>MULTIPLY (64&lt;32)</td>
<td>M</td>
<td>RX C SP</td>
<td>Bz 7-148</td>
</tr>
<tr>
<td>5D</td>
<td>DIVIDE (32&lt;64)</td>
<td>D</td>
<td>RX A SP IK</td>
<td>Bz 7-116</td>
</tr>
<tr>
<td>5E</td>
<td>ADD LOGICAL (32)</td>
<td>AL</td>
<td>RX C A</td>
<td>Bz 7-19</td>
</tr>
<tr>
<td>5F</td>
<td>SUBTRACT LOGICAL (32)</td>
<td>SL</td>
<td>RX C A</td>
<td>Bz 7-189</td>
</tr>
<tr>
<td>60</td>
<td>STORE (long)</td>
<td>STD</td>
<td>RX A Da ST</td>
<td>Bz 9-11</td>
</tr>
<tr>
<td>61</td>
<td>STORE (short)</td>
<td>STE</td>
<td>RX A Da ST</td>
<td>Bz 9-11</td>
</tr>
<tr>
<td>67</td>
<td>MULTIPLY (long to extended HFP)</td>
<td>MXD</td>
<td>RX A SP Da EU EO</td>
<td>Bz 18-18</td>
</tr>
<tr>
<td>68</td>
<td>LOAD (long)</td>
<td>LD</td>
<td>RX A Da</td>
<td>Bz 9-10</td>
</tr>
<tr>
<td>69</td>
<td>COMPARE (long HFP)</td>
<td>CD</td>
<td>RX C A Da</td>
<td>Bz 18-10</td>
</tr>
<tr>
<td>6A</td>
<td>ADD NORMALIZED (long HFP)</td>
<td>AD</td>
<td>RX C A Da EU EO LS</td>
<td>Bz 18-8</td>
</tr>
<tr>
<td>6B</td>
<td>SUBTRACT NORMALIZED (long HFP)</td>
<td>SD</td>
<td>RX C A Da EU EO LS</td>
<td>Bz 18-22</td>
</tr>
<tr>
<td>6C</td>
<td>MULTIPLY (long HFP)</td>
<td>MD</td>
<td>RX A Da EU EO</td>
<td>Bz 18-18</td>
</tr>
<tr>
<td>6D</td>
<td>DIVIDE (long HFP)</td>
<td>DD</td>
<td>RX A Da EU EO FK</td>
<td>Bz 18-12</td>
</tr>
<tr>
<td>6E</td>
<td>ADD UNNORMALIZED (long HFP)</td>
<td>AW</td>
<td>RX C A Da EO LS</td>
<td>Bz 18-10</td>
</tr>
<tr>
<td>6F</td>
<td>SUBTRACT UNNORMALIZED (long HFP)</td>
<td>SW</td>
<td>RX C A Da EO LS</td>
<td>Bz 18-22</td>
</tr>
<tr>
<td>70</td>
<td>STORE (short)</td>
<td>STE</td>
<td>RX A Da ST</td>
<td>Bz 9-11</td>
</tr>
<tr>
<td>71</td>
<td>MULTIPLY SINGLE (32)</td>
<td>MS</td>
<td>RX A</td>
<td>Bz 7-151</td>
</tr>
<tr>
<td>78</td>
<td>LOAD (short)</td>
<td>LE</td>
<td>RX A Da</td>
<td>Bz 9-10</td>
</tr>
<tr>
<td>79</td>
<td>COMPARE (short HFP)</td>
<td>CE</td>
<td>RX C A Da</td>
<td>Bz 18-10</td>
</tr>
<tr>
<td>7A</td>
<td>ADD NORMALIZED (short HFP)</td>
<td>AE</td>
<td>RX C A Da EU EO LS</td>
<td>Bz 18-8</td>
</tr>
<tr>
<td>7B</td>
<td>SUBTRACT NORMALIZED (short HFP)</td>
<td>SE</td>
<td>RX C A Da EU EO LS</td>
<td>Bz 18-22</td>
</tr>
<tr>
<td>7C</td>
<td>MULTIPLY (short to long HFP)</td>
<td>MDE</td>
<td>RX A Da EU EO</td>
<td>Bz 18-18</td>
</tr>
<tr>
<td>7D</td>
<td>DIVIDE (short HFP)</td>
<td>DED</td>
<td>RX A Da EU EO FK</td>
<td>Bz 18-12</td>
</tr>
<tr>
<td>7E</td>
<td>ADD UNNORMALIZED (short HFP)</td>
<td>AU</td>
<td>RX C A Da EO LS</td>
<td>Bz 18-10</td>
</tr>
<tr>
<td>7F</td>
<td>SUBTRACT UNNORMALIZED (short HFP)</td>
<td>SU</td>
<td>RX C A Da EO LS</td>
<td>Bz 18-22</td>
</tr>
<tr>
<td>80</td>
<td>SET SYSTEM MASK</td>
<td>SSM</td>
<td>S P A SP SO</td>
<td>Bz 18-105</td>
</tr>
<tr>
<td>82</td>
<td>LOAD PSW</td>
<td>LPSW</td>
<td>S L P A SP</td>
<td>Bz 10-45</td>
</tr>
<tr>
<td>83</td>
<td>DIAGNOSE</td>
<td>DM</td>
<td>P A SP δ</td>
<td>MD 10-19</td>
</tr>
<tr>
<td>84</td>
<td>BRANCH RELATIVE ON INDEX HIGH (32)</td>
<td>BRXH</td>
<td>RXI P A SP DM</td>
<td>B 7-30</td>
</tr>
<tr>
<td>85</td>
<td>BRANCH RELATIVE ON INDEX L OR E (32)</td>
<td>BRXLLE</td>
<td>RXI P A SP DM</td>
<td>B 7-30</td>
</tr>
<tr>
<td>86</td>
<td>BRANCH ON INDEX HIGH (32)</td>
<td>BXH</td>
<td>RS P A SP</td>
<td>B 7-27</td>
</tr>
<tr>
<td>87</td>
<td>BRANCH ON INDEX LOW OR EQUAL (32)</td>
<td>BXLLE</td>
<td>RS P A SP</td>
<td>B 7-27</td>
</tr>
<tr>
<td>88</td>
<td>SHIFT RIGHT SINGLE LOGICAL (32)</td>
<td>SRL</td>
<td>RS C</td>
<td>B 7-180</td>
</tr>
<tr>
<td>89</td>
<td>SHIFT LEFT SINGLE LOGICAL (32)</td>
<td>SLL</td>
<td>RS C</td>
<td>B 7-178</td>
</tr>
<tr>
<td>8A</td>
<td>SHIFT RIGHT SINGLE (32)</td>
<td>SRA</td>
<td>RS C</td>
<td>B 7-179</td>
</tr>
<tr>
<td>8B</td>
<td>SHIFT LEFT SINGLE (32)</td>
<td>SLA</td>
<td>RS C IF</td>
<td>7-177</td>
</tr>
<tr>
<td>8C</td>
<td>SHIFT RIGHT DOUBLE LOGICAL</td>
<td>SRDL</td>
<td>RS SP</td>
<td>7-179</td>
</tr>
<tr>
<td>8D</td>
<td>SHIFT LEFT DOUBLE LOGICAL</td>
<td>SLDL</td>
<td>RS SP</td>
<td>7-176</td>
</tr>
<tr>
<td>8E</td>
<td>SHIFT RIGHT DOUBLE</td>
<td>SRDA</td>
<td>RS C SP</td>
<td>7-178</td>
</tr>
<tr>
<td>8F</td>
<td>SHIFT LEFT DOUBLE</td>
<td>SLDA</td>
<td>RS C SP IF</td>
<td>7-176</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 3 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>STORE MULTIPLE (32)</td>
<td>STM</td>
<td>RS A</td>
<td>ST B2</td>
</tr>
<tr>
<td>91</td>
<td>TEST UNDER MASK</td>
<td>TM</td>
<td>SI C A</td>
<td>ST B1</td>
</tr>
<tr>
<td>92</td>
<td>MOVE (immediate)</td>
<td>MVI</td>
<td>SI A</td>
<td>ST B1</td>
</tr>
<tr>
<td>93</td>
<td>TEST AND SET</td>
<td>TS</td>
<td>S C A $</td>
<td>ST B2</td>
</tr>
<tr>
<td>94</td>
<td>AND (immediate)</td>
<td>NI</td>
<td>SI C A</td>
<td>ST B1</td>
</tr>
<tr>
<td>95</td>
<td>COMPARISON LOGICAL (immediate)</td>
<td>CLI</td>
<td>SI C A</td>
<td>ST B1</td>
</tr>
<tr>
<td>96</td>
<td>OR (immediate)</td>
<td>OI</td>
<td>SI C A</td>
<td>ST B1</td>
</tr>
<tr>
<td>97</td>
<td>EXCLUSIVE OR (immediate)</td>
<td>XI</td>
<td>SI C A</td>
<td>ST B2</td>
</tr>
<tr>
<td>98</td>
<td>LOAD MULTIPLE (32)</td>
<td>LM</td>
<td>RS A</td>
<td>ST B2</td>
</tr>
<tr>
<td>99</td>
<td>TRACE (32)</td>
<td>TRACE</td>
<td>RS P A SP T ɛ</td>
<td>ST B2</td>
</tr>
<tr>
<td>9A</td>
<td>LOAD ACCESS MULTIPLE</td>
<td>LAM</td>
<td>RS A SP</td>
<td>ST B2</td>
</tr>
<tr>
<td>9B</td>
<td>STORE ACCESS MULTIPLE</td>
<td>STAM</td>
<td>RS A SP</td>
<td>ST B2</td>
</tr>
<tr>
<td>A50</td>
<td>INSERT IMMEDIATE (high high)</td>
<td>IHHH</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A51</td>
<td>INSERT IMMEDIATE (high low)</td>
<td>IHHL</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A52</td>
<td>INSERT IMMEDIATE (low high)</td>
<td>IILH</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A53</td>
<td>INSERT IMMEDIATE (low low)</td>
<td>IILL</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A54</td>
<td>AND IMMEDIATE (high high)</td>
<td>NIHH</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A55</td>
<td>AND IMMEDIATE (high low)</td>
<td>NIH</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A56</td>
<td>AND IMMEDIATE (low high)</td>
<td>NIL</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A57</td>
<td>AND IMMEDIATE (low low)</td>
<td>NILL</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A58</td>
<td>OR IMMEDIATE (high high)</td>
<td>OIHH</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A59</td>
<td>OR IMMEDIATE (high low)</td>
<td>OIHL</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A5A</td>
<td>OR IMMEDIATE (low high)</td>
<td>OILH</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A5B</td>
<td>OR IMMEDIATE (low low)</td>
<td>OILL</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A5C</td>
<td>LOAD LOGICAL IMMEDIATE (high high)</td>
<td>LLIHH</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A5D</td>
<td>LOAD LOGICAL IMMEDIATE (high low)</td>
<td>LLIH</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A5E</td>
<td>LOAD LOGICAL IMMEDIATE (low high)</td>
<td>LLLH</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A5F</td>
<td>LOAD LOGICAL IMMEDIATE (low low)</td>
<td>LLL</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A70</td>
<td>TEST UNDER MASK (low high)</td>
<td>TMLH</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A71</td>
<td>TEST UNDER MASK (low low)</td>
<td>TMLL</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A72</td>
<td>TEST UNDER MASK (high high)</td>
<td>TMHH</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A73</td>
<td>TEST UNDER MASK (high low)</td>
<td>TMHL</td>
<td>RI C N</td>
<td>UB</td>
</tr>
<tr>
<td>A74</td>
<td>BRANCH RELATIVE ON CONDITION</td>
<td>BRC</td>
<td>RI</td>
<td>UB</td>
</tr>
<tr>
<td>A75</td>
<td>BRANCH RELATIVE AND SAVE</td>
<td>BRAS</td>
<td>RI</td>
<td>UB</td>
</tr>
<tr>
<td>A76</td>
<td>BRANCH RELATIVE ON COUNT (32)</td>
<td>BRCT</td>
<td>RI</td>
<td>UB</td>
</tr>
<tr>
<td>A77</td>
<td>BRANCH RELATIVE ON COUNT (64)</td>
<td>BRCTG</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A78</td>
<td>LOAD HALFWORD IMMEDIATE (32)</td>
<td>LHI</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A79</td>
<td>LOAD HALFWORD IMMEDIATE (64)</td>
<td>LGHI</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A7A</td>
<td>ADD HALFWORD IMMEDIATE (32)</td>
<td>AHI</td>
<td>RI C IF</td>
<td>UB</td>
</tr>
<tr>
<td>A7B</td>
<td>ADD HALFWORD IMMEDIATE (64)</td>
<td>AGHI</td>
<td>RI C IF</td>
<td>UB</td>
</tr>
<tr>
<td>A7C</td>
<td>MULTIPLY HALFWORD IMMEDIATE (32)</td>
<td>MHI</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A7D</td>
<td>MULTIPLY HALFWORD IMMEDIATE (64)</td>
<td>MGHI</td>
<td>RI N</td>
<td>UB</td>
</tr>
<tr>
<td>A7E</td>
<td>COMPARE HALFWORD IMMEDIATE (32)</td>
<td>CHI</td>
<td>RI C</td>
<td>UB</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 4 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7F</td>
<td>COM-</td>
<td>CGHI</td>
<td>RI C N</td>
<td>7-55</td>
</tr>
<tr>
<td>A8</td>
<td>MOVe</td>
<td>MVCLE</td>
<td>RS C A SP</td>
<td>7-139</td>
</tr>
<tr>
<td>A9</td>
<td>COM-</td>
<td>CLCLE</td>
<td>RS C A SP</td>
<td>7-60</td>
</tr>
<tr>
<td>AC</td>
<td>STORE</td>
<td>STNSM</td>
<td>SI P A</td>
<td>10-124</td>
</tr>
<tr>
<td>AD</td>
<td>STORE</td>
<td>STOSM</td>
<td>SI P A SP</td>
<td>10-124</td>
</tr>
<tr>
<td>AE</td>
<td>SIGNAL</td>
<td>SIGP</td>
<td>RS C P $</td>
<td>10-106</td>
</tr>
<tr>
<td>AF</td>
<td>MONI-</td>
<td>MC</td>
<td>SI SP MO</td>
<td>7-133</td>
</tr>
<tr>
<td>B1</td>
<td>LOAD</td>
<td>LRA</td>
<td>RX C P A SP SO</td>
<td>BP Bz</td>
</tr>
<tr>
<td>B202</td>
<td>CPU</td>
<td>STIDP</td>
<td>S P A SP</td>
<td>Bz</td>
</tr>
<tr>
<td>B204</td>
<td>SET</td>
<td>SCK</td>
<td>S C P A SP</td>
<td>Bz</td>
</tr>
<tr>
<td>B205</td>
<td>STORE</td>
<td>STCK</td>
<td>S C A $</td>
<td>Bz</td>
</tr>
<tr>
<td>B206</td>
<td>STORE</td>
<td>SCKC</td>
<td>S P A SP</td>
<td>Bz</td>
</tr>
<tr>
<td>B207</td>
<td>STORE</td>
<td>STCKC</td>
<td>S P A SP</td>
<td>Bz</td>
</tr>
<tr>
<td>B208</td>
<td>SET</td>
<td>SPT</td>
<td>S P A SP</td>
<td>Bz</td>
</tr>
<tr>
<td>B209</td>
<td>STORE</td>
<td>STPT</td>
<td>S P A SP</td>
<td>Bz</td>
</tr>
<tr>
<td>B20A</td>
<td>SET</td>
<td>SPKA</td>
<td>S Q G2</td>
<td>10-99</td>
</tr>
<tr>
<td>B20B</td>
<td>INSERT</td>
<td>IPK</td>
<td>S Q $</td>
<td>10-28</td>
</tr>
<tr>
<td>B20D</td>
<td>PURGE</td>
<td>PTLB</td>
<td>S P $</td>
<td>10-91</td>
</tr>
<tr>
<td>B210</td>
<td>SET</td>
<td>SPX</td>
<td>S P A SP $</td>
<td>Bz</td>
</tr>
<tr>
<td>B211</td>
<td>STORE</td>
<td>STPX</td>
<td>S P A SP</td>
<td>Bz</td>
</tr>
<tr>
<td>B212</td>
<td>STORE</td>
<td>STAP</td>
<td>S P A SP</td>
<td>ST Bz</td>
</tr>
<tr>
<td>B218</td>
<td>PROGRAM</td>
<td>PC</td>
<td>S Q A1 Z1 T ́ GM</td>
<td>Bz</td>
</tr>
<tr>
<td>B219</td>
<td>SET</td>
<td>SAC</td>
<td>S Q SP SW ́</td>
<td>10-95</td>
</tr>
<tr>
<td>B21A</td>
<td>COMPARE</td>
<td>CFC</td>
<td>S C A SP II GM</td>
<td>I1 7-46</td>
</tr>
<tr>
<td>B221</td>
<td>INVALIDATE</td>
<td>IPTE</td>
<td>RRE P A $</td>
<td>10-110</td>
</tr>
<tr>
<td>B222</td>
<td>INSERT</td>
<td>IPM</td>
<td>RRE Q A1 SO</td>
<td>7-122</td>
</tr>
<tr>
<td>B223</td>
<td>INSERT</td>
<td>IVSK</td>
<td>RRE Q A1 SO</td>
<td>Rz</td>
</tr>
<tr>
<td>B224</td>
<td>INSERT</td>
<td>IAC</td>
<td>RRE Q SO</td>
<td>10-27</td>
</tr>
<tr>
<td>B225</td>
<td>SET</td>
<td>SSAR</td>
<td>RRE A1 Z2 T ́</td>
<td>10-99</td>
</tr>
<tr>
<td>B226</td>
<td>EXTRACT</td>
<td>EPAR</td>
<td>RRE Q</td>
<td>10-21</td>
</tr>
<tr>
<td>B227</td>
<td>EXTRACT</td>
<td>ESAR</td>
<td>RRE Q A1 SO</td>
<td>10-22</td>
</tr>
<tr>
<td>B228</td>
<td>PROGRAM</td>
<td>PT</td>
<td>RRE Q A1 SP</td>
<td>10-81</td>
</tr>
<tr>
<td>B229</td>
<td>INSERT</td>
<td>ISKE</td>
<td>RRE P A1</td>
<td>10-28</td>
</tr>
<tr>
<td>B22A</td>
<td>RESET</td>
<td>RRBE</td>
<td>RRE A1 P A1 ́</td>
<td>10-91</td>
</tr>
<tr>
<td>B22B</td>
<td>SET STORAGE</td>
<td>SSKE</td>
<td>RRE P A1 ́</td>
<td>10-105</td>
</tr>
<tr>
<td>B22C</td>
<td>TEST</td>
<td>TB</td>
<td>RRE A1 II $ G0</td>
<td>10-127</td>
</tr>
<tr>
<td>B22D</td>
<td>DIVIDE</td>
<td>DXR</td>
<td>RRE A1 SP Da EU EO FK</td>
<td>10-12</td>
</tr>
<tr>
<td>B22E</td>
<td>PAGE</td>
<td>PGIN</td>
<td>RRE C ES P A1</td>
<td>10-61</td>
</tr>
<tr>
<td>B22F</td>
<td>PAGE</td>
<td>PGOUT</td>
<td>RRE C ES P A1 ́</td>
<td>10-62</td>
</tr>
<tr>
<td>B230</td>
<td>CLEAR</td>
<td>CSCH</td>
<td>C P OP ́ GS</td>
<td>14-5</td>
</tr>
<tr>
<td>B231</td>
<td>HALT</td>
<td>HSCH</td>
<td>S C P OP ́ GS</td>
<td>14-6</td>
</tr>
<tr>
<td>B232</td>
<td>MODIFY</td>
<td>MSCH</td>
<td>S C P A SP OP ́ GS</td>
<td>Bz 14-7</td>
</tr>
<tr>
<td>B233</td>
<td>START</td>
<td>SSCH</td>
<td>S C P A SP OP ́ GS</td>
<td>Bz 14-14</td>
</tr>
<tr>
<td>B234</td>
<td>STORE</td>
<td>STSCH</td>
<td>S C P A SP OP ́ GS</td>
<td>ST Bz 14-17</td>
</tr>
<tr>
<td>B235</td>
<td>TEST</td>
<td>TSCH</td>
<td>S C P A SP OP ́ GS</td>
<td>ST Bz 14-20</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 5 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B236</td>
<td>TEST PENDING INTERRUPTION</td>
<td>TPI</td>
<td>P A1 SP</td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td>B237</td>
<td>SET ADDRESS LIMIT</td>
<td>SAL</td>
<td>P</td>
<td>OP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G1</td>
</tr>
<tr>
<td>B238</td>
<td>RESUME SUBCHANNEL</td>
<td>RSCH</td>
<td>C</td>
<td>P</td>
</tr>
<tr>
<td>B239</td>
<td>STORE CHANNEL REPORT WORD</td>
<td>STCRW</td>
<td>C</td>
<td>P A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td>B23A</td>
<td>STORE CHANNEL PATH STATUS</td>
<td>STCPS</td>
<td>C</td>
<td>P A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B2</td>
</tr>
<tr>
<td>B23B</td>
<td>RESET CHANNEL PATH</td>
<td>RCHP</td>
<td>C</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G1</td>
</tr>
<tr>
<td>B23C</td>
<td>SET CHANNEL MONITOR</td>
<td>SCHM</td>
<td>C</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>G M</td>
</tr>
<tr>
<td>B240</td>
<td>BRANCH AND STACK</td>
<td>BAKR</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B241</td>
<td>CHECKSUM</td>
<td>C KSM</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B242</td>
<td>SQUARE ROOT (long HFP)</td>
<td>SQDR</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B243</td>
<td>SQUARE ROOT (short HFP)</td>
<td>STURA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B244</td>
<td>STORE USING REAL ADDRESS (32)</td>
<td>STURA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B245</td>
<td>STORE USING REAL ADDRESS (32)</td>
<td>STURA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B246</td>
<td>STORE ADDRESS LIMIT</td>
<td>STURA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B247</td>
<td>MODIFY STACKED STATE</td>
<td>MSTA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B248</td>
<td>PURGE ALB</td>
<td>PALB</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B249</td>
<td>EXTRACT STACKED REGISTERS (32)</td>
<td>P E R E G</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B24A</td>
<td>EXTRACT STACKED STATE</td>
<td>ESTA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B24B</td>
<td>LOAD USING REAL ADDRESS (32)</td>
<td>LURA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B24C</td>
<td>TEST ACCESS</td>
<td>TAR</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B24D</td>
<td>COPY ACCESS</td>
<td>CPYA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B24E</td>
<td>SET ACCESS</td>
<td>SAR</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B24F</td>
<td>EXTRACT ACCESS</td>
<td>EAR</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B250</td>
<td>COMPARE AND SWAP AND PURGE</td>
<td>CSP</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B254</td>
<td>MOVE PAGE</td>
<td>MVPG</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B255</td>
<td>MOVE STRING</td>
<td>MVST</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B257</td>
<td>COMPARISON UNTIL SUBSTRING EQUAL</td>
<td>USE</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B258</td>
<td>BRANCH IN SUBSPACE GROUP</td>
<td>BSG</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B259</td>
<td>BRANCH AND SET AUTHORITY</td>
<td>BSA</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B25D</td>
<td>COMPARE LOGICAL STRING</td>
<td>CLST</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B25E</td>
<td>SEARCH STRING</td>
<td>SRST</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B263</td>
<td>COMPRESS CALL</td>
<td>CMPSC</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B265</td>
<td>STORE FACILITY LIST</td>
<td>STFL</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B267</td>
<td>CANCEL SUBCHANNEL</td>
<td>X SCH</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B277</td>
<td>RESUME PROGRAM</td>
<td>RP</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B278</td>
<td>STORE CLOCK EXTENDED</td>
<td>STCKE</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B279</td>
<td>SET ADDRESS SPACE CONTROL FAST</td>
<td>STCF</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B270</td>
<td>STORE SYSTEM INFORMATION</td>
<td>STSI</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B299</td>
<td>SET Rounding MODE</td>
<td>SRNM</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B290</td>
<td>STORE FPC</td>
<td>STFPC</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B291</td>
<td>LOAD FPC</td>
<td>L FPC</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B2A5</td>
<td>TRANSLATE EXTENDED</td>
<td>TRE</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B2A6</td>
<td>CONVERT UNICODE TO UTF-8</td>
<td>CUUF</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B2A7</td>
<td>CONVERT UTF-8 TO UNICODE</td>
<td>CU F</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B2A8</td>
<td>CONVERT UTF-8 TO UTF-16</td>
<td>CU I</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
<tr>
<td>B2A9</td>
<td>STORE FACILITY LIST</td>
<td>STFL</td>
<td>RRE</td>
<td>A SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SQ</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 6 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mne-monic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3B2</td>
<td>LOAD PSW EXTENDED</td>
<td>LPSWE</td>
<td>S L N P A SP T</td>
<td>10-46</td>
</tr>
<tr>
<td>B3FF</td>
<td>TRAP</td>
<td>TRAP4</td>
<td>S</td>
<td>10-133</td>
</tr>
<tr>
<td>B300</td>
<td>LOAD POSITIVE (short BFP)</td>
<td>LPEBR</td>
<td>RRE C</td>
<td>19-38</td>
</tr>
<tr>
<td>B301</td>
<td>LOAD NEGATIVE (short BFP)</td>
<td>LNEBR</td>
<td>RRE C</td>
<td>19-37</td>
</tr>
<tr>
<td>B302</td>
<td>LOAD AND TEST (short BFP)</td>
<td>LTEBR</td>
<td>RRE C</td>
<td>19-34</td>
</tr>
<tr>
<td>B303</td>
<td>LOAD COMPLEMENT (short BFP)</td>
<td>LCEBR</td>
<td>RRE C</td>
<td>19-34</td>
</tr>
<tr>
<td>B304</td>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>LDEBR</td>
<td>RRE</td>
<td>19-37</td>
</tr>
<tr>
<td>B305</td>
<td>LOAD LENGTHENED (long to ext. BFP)</td>
<td>LXDBR</td>
<td>RRE SP</td>
<td>19-37</td>
</tr>
<tr>
<td>B306</td>
<td>LOAD LENGTHENED (short to ext. BFP)</td>
<td>LXEBR</td>
<td>RRE SP</td>
<td>19-37</td>
</tr>
<tr>
<td>B307</td>
<td>MULTIPLY (long to extended BFP)</td>
<td>MXDBR</td>
<td>RRE</td>
<td>19-39</td>
</tr>
<tr>
<td>B308</td>
<td>COMPARE AND SIGNAL (short BFP)</td>
<td>KEBR</td>
<td>RRE C</td>
<td>19-24</td>
</tr>
<tr>
<td>B309</td>
<td>COMPARE (short BFP)</td>
<td>CEBR</td>
<td>RRE C</td>
<td>19-23</td>
</tr>
<tr>
<td>B30A</td>
<td>ADD (short BFP)</td>
<td>AEBR</td>
<td>RRE C</td>
<td>19-18</td>
</tr>
<tr>
<td>B30B</td>
<td>SUBTRACT (short BFP)</td>
<td>SEBR</td>
<td>RRE C</td>
<td>19-44</td>
</tr>
<tr>
<td>B30C</td>
<td>MULTIPLY (short to long BFP)</td>
<td>MDEBR</td>
<td>RRE</td>
<td>19-39</td>
</tr>
<tr>
<td>B30D</td>
<td>DIVIDE (short BFP)</td>
<td>DEBR</td>
<td>RRE</td>
<td>19-29</td>
</tr>
<tr>
<td>B30E</td>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>MAEBR</td>
<td>RRF</td>
<td>19-41</td>
</tr>
<tr>
<td>B30F</td>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSEBR</td>
<td>RRF</td>
<td>19-41</td>
</tr>
<tr>
<td>B310</td>
<td>LOAD POSITIVE (long BFP)</td>
<td>LPDBR</td>
<td>RRE C</td>
<td>19-38</td>
</tr>
<tr>
<td>B311</td>
<td>LOAD NEGATIVE (long BFP)</td>
<td>LNDDBR</td>
<td>RRE C</td>
<td>19-37</td>
</tr>
<tr>
<td>B312</td>
<td>LOAD AND TEST (long BFP)</td>
<td>LTDBR</td>
<td>RRE C</td>
<td>19-34</td>
</tr>
<tr>
<td>B313</td>
<td>LOAD COMPLEMENT (long BFP)</td>
<td>LCDDBR</td>
<td>RRE</td>
<td>19-34</td>
</tr>
<tr>
<td>B314</td>
<td>SQUARE ROOT (short BFP)</td>
<td>SQEBR</td>
<td>RRE</td>
<td>19-44</td>
</tr>
<tr>
<td>B315</td>
<td>SQUARE ROOT (long BFP)</td>
<td>SQDBR</td>
<td>RRE</td>
<td>19-44</td>
</tr>
<tr>
<td>B316</td>
<td>SQUARE ROOT (extended BFP)</td>
<td>SQXBR</td>
<td>SP</td>
<td>19-44</td>
</tr>
<tr>
<td>B317</td>
<td>MULTIPLY (short BFP)</td>
<td>MEEBR</td>
<td>RRE</td>
<td>19-39</td>
</tr>
<tr>
<td>B318</td>
<td>COMPARE AND SIGNAL (long BFP)</td>
<td>KDBR</td>
<td>RRE C</td>
<td>19-24</td>
</tr>
<tr>
<td>B319</td>
<td>COMPARE (long BFP)</td>
<td>CDBR</td>
<td>RRE C</td>
<td>19-23</td>
</tr>
<tr>
<td>B31A</td>
<td>ADD (long BFP)</td>
<td>ADDBR</td>
<td>RRE C</td>
<td>19-18</td>
</tr>
<tr>
<td>B31B</td>
<td>SUBTRACT (long BFP)</td>
<td>SDBR</td>
<td>RRE C</td>
<td>19-44</td>
</tr>
<tr>
<td>B31C</td>
<td>MULTIPLY (long BFP)</td>
<td>MDBR</td>
<td>RRE</td>
<td>19-39</td>
</tr>
<tr>
<td>B31D</td>
<td>DIVIDE (long BFP)</td>
<td>DDBR</td>
<td>RRE</td>
<td>19-29</td>
</tr>
<tr>
<td>B31E</td>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>MADBR</td>
<td>RRF</td>
<td>19-41</td>
</tr>
<tr>
<td>B31F</td>
<td>MULTIPLY AND SUBTRACT (long BFP)</td>
<td>MSDBR</td>
<td>RRF</td>
<td>19-41</td>
</tr>
<tr>
<td>B324</td>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>LDER</td>
<td>RRE</td>
<td>18-15</td>
</tr>
<tr>
<td>B325</td>
<td>LOAD LENGTHENED (long to ext. HFP)</td>
<td>LXDR</td>
<td>RRE SP</td>
<td>18-15</td>
</tr>
<tr>
<td>B326</td>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>LXER</td>
<td>RRE SP</td>
<td>18-15</td>
</tr>
<tr>
<td>B32E</td>
<td>MULTIPLY AND ADD (short HFP)</td>
<td>MAER</td>
<td>RRF HM</td>
<td>18-19</td>
</tr>
<tr>
<td>B32F</td>
<td>MULTIPLY AND SUBTRACT (short HFP)</td>
<td>MSER</td>
<td>RRF HM</td>
<td>18-20</td>
</tr>
<tr>
<td>B336</td>
<td>SQUARE ROOT (extended HFP)</td>
<td>SQXBR</td>
<td>RRE SP</td>
<td>18-21</td>
</tr>
<tr>
<td>B337</td>
<td>MULTIPLY (short HFP)</td>
<td>MEER</td>
<td>RRE</td>
<td>18-18</td>
</tr>
<tr>
<td>B33E</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>MADR</td>
<td>RRF HM</td>
<td>18-19</td>
</tr>
<tr>
<td>B33F</td>
<td>MULTIPLY AND SUBTRACT (long HFP)</td>
<td>MSDR</td>
<td>RRF HM</td>
<td>18-20</td>
</tr>
<tr>
<td>B340</td>
<td>LOAD POSITIVE (extended BFP)</td>
<td>LPSBR</td>
<td>RRE SP</td>
<td>19-38</td>
</tr>
<tr>
<td>B341</td>
<td>LOAD NEGATIVE (extended BFP)</td>
<td>LNXBR</td>
<td>RRE SP</td>
<td>19-37</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 7 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B342</td>
<td>LOAD AND TEST (extended BFP)</td>
<td>LTXBR</td>
<td>RRE C SP Db Xi</td>
<td>19-34</td>
</tr>
<tr>
<td>B343</td>
<td>LOAD COMPLEMENT (extended BFP)</td>
<td>LCXBR</td>
<td>RRE C SP Db</td>
<td>19-34</td>
</tr>
<tr>
<td>B344</td>
<td>LOAD ROUNDED (long to short BFP)</td>
<td>LEDBR</td>
<td>RRE SP Db Xi Xo Xu Xx</td>
<td>19-38</td>
</tr>
<tr>
<td>B345</td>
<td>LOAD ROUNDED (extended to long BFP)</td>
<td>LXDR</td>
<td>RRE SP Db Xi Xo Xu Xx</td>
<td>19-38</td>
</tr>
<tr>
<td>B346</td>
<td>LOAD ROUNDED (extended to short BFP)</td>
<td>LEXBR</td>
<td>RRE SP Db Xi Xo Xu Xx</td>
<td>19-38</td>
</tr>
<tr>
<td>B347</td>
<td>LOAD FP INTEGER (extended BFP)</td>
<td>FIBR</td>
<td>RRE SP Db Xi Xx</td>
<td>19-35</td>
</tr>
<tr>
<td>B348</td>
<td>COMPARE AND SIGNAL (extended BFP)</td>
<td>KXBR</td>
<td>RRE C SP Db Xi</td>
<td>19-24</td>
</tr>
<tr>
<td>B349</td>
<td>COMPARE (extended BFP)</td>
<td>CXBR</td>
<td>RRE C SP Db Xi</td>
<td>19-23</td>
</tr>
<tr>
<td>B34A</td>
<td>ADD (extended BFP)</td>
<td>AXBR</td>
<td>RRE C SP Db Xi Xo Xu Xx</td>
<td>19-18</td>
</tr>
<tr>
<td>B34B</td>
<td>SUBTRACT (extended BFP)</td>
<td>SXBR</td>
<td>RRE C SP Db Xi Xo Xu Xx</td>
<td>19-44</td>
</tr>
<tr>
<td>B34C</td>
<td>MULTIPLY (extended BFP)</td>
<td>MIBR</td>
<td>RRE SP Db Xi Xo Xu Xx</td>
<td>19-39</td>
</tr>
<tr>
<td>B34D</td>
<td>DIVIDE (extended BFP)</td>
<td>DIBR</td>
<td>RRE SP Db Xi Xz Xo Xu Xx</td>
<td>19-29</td>
</tr>
<tr>
<td>B350</td>
<td>CONVERT HFP TO BFP (long to short)</td>
<td>TBEDR</td>
<td>RRE C SP Da</td>
<td>9-9</td>
</tr>
<tr>
<td>B351</td>
<td>CONVERT HFP TO BFP (long)</td>
<td>TBRD</td>
<td>RRE C SP Da</td>
<td>9-9</td>
</tr>
<tr>
<td>B353</td>
<td>DIVIDE TO INTEGER (short BFP)</td>
<td>DIBR</td>
<td>RRE C SP Da</td>
<td>19-29</td>
</tr>
<tr>
<td>B357</td>
<td>LOAD FP INTEGER (short BFP)</td>
<td>FIBR</td>
<td>RRE SP Db Xi Xx</td>
<td>19-35</td>
</tr>
<tr>
<td>B358</td>
<td>CONVERT BFP TO HFP (short to long)</td>
<td>THEDR</td>
<td>RRE C SP Da</td>
<td>9-8</td>
</tr>
<tr>
<td>B359</td>
<td>CONVERT BFP TO HFP (long)</td>
<td>TBRD</td>
<td>RRE C SP Da</td>
<td>9-8</td>
</tr>
<tr>
<td>B35B</td>
<td>DIVIDE TO INTEGER (long BFP)</td>
<td>DIBR</td>
<td>RRE C SP Da</td>
<td>19-29</td>
</tr>
<tr>
<td>B35F</td>
<td>LOAD FP INTEGER (long BFP)</td>
<td>FIBR</td>
<td>RRE SP Db Xi Xx</td>
<td>19-35</td>
</tr>
<tr>
<td>B360</td>
<td>LOAD POSITIVE (extended HFP)</td>
<td>LPXR</td>
<td>RRE C SP Da</td>
<td>18-16</td>
</tr>
<tr>
<td>B361</td>
<td>LOAD NEGATIVE (extended HFP)</td>
<td>LNXR</td>
<td>RRE C SP Da</td>
<td>18-16</td>
</tr>
<tr>
<td>B362</td>
<td>LOAD AND TEST (extended HFP)</td>
<td>LXBR</td>
<td>RRE C SP Da</td>
<td>18-14</td>
</tr>
<tr>
<td>B363</td>
<td>LOAD COMPLEMENT (extended HFP)</td>
<td>LCXR</td>
<td>RRE C SP Da</td>
<td>18-14</td>
</tr>
<tr>
<td>B365</td>
<td>LOAD (extended)</td>
<td>LXR</td>
<td>RRE SP Da</td>
<td>9-10</td>
</tr>
<tr>
<td>B366</td>
<td>LOAD ROUNDED (extended to short HFP)</td>
<td>LEXR</td>
<td>RRE SP Da E0</td>
<td>18-17</td>
</tr>
<tr>
<td>B367</td>
<td>LOAD FP INTEGER (extended HFP)</td>
<td>FIXR</td>
<td>RRE SP Da</td>
<td>18-15</td>
</tr>
<tr>
<td>B369</td>
<td>COMPARE (extended HFP)</td>
<td>CXR</td>
<td>RRE C SP Da</td>
<td>18-10</td>
</tr>
<tr>
<td>B374</td>
<td>LOAD ZERO (short)</td>
<td>LZR</td>
<td>RRE SP Da</td>
<td>9-11</td>
</tr>
<tr>
<td>B375</td>
<td>LOAD ZERO (long)</td>
<td>LZDR</td>
<td>RRE SP Da</td>
<td>9-11</td>
</tr>
<tr>
<td>B376</td>
<td>LOAD ZERO (extended)</td>
<td>LZXR</td>
<td>RRE SP Da</td>
<td>9-11</td>
</tr>
<tr>
<td>B377</td>
<td>LOAD FP INTEGER (short HFP)</td>
<td>FIER</td>
<td>RRE SP Da</td>
<td>18-15</td>
</tr>
<tr>
<td>B37F</td>
<td>LOAD FP INTEGER (long HFP)</td>
<td>FIDR</td>
<td>RRE SP Da</td>
<td>18-15</td>
</tr>
<tr>
<td>B384</td>
<td>SET FPC</td>
<td>SFPC</td>
<td>RRE SP Db</td>
<td>19-43</td>
</tr>
<tr>
<td>B38C</td>
<td>EXTRACT FPC</td>
<td>EFPC</td>
<td>RRE SP Db</td>
<td>19-33</td>
</tr>
<tr>
<td>B394</td>
<td>CONVERT FROM FIXED (32 to short BFP)</td>
<td>CEFBR</td>
<td>RRE SP Db Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B395</td>
<td>CONVERT FROM FIXED (32 to long BFP)</td>
<td>CFBR</td>
<td>RRE SP Db</td>
<td>19-26</td>
</tr>
<tr>
<td>B396</td>
<td>CONVERT FROM FIXED (32 to ext. BFP)</td>
<td>CFXBR</td>
<td>RRE SP Db</td>
<td>19-26</td>
</tr>
<tr>
<td>B398</td>
<td>CONVERT TO FIXED (short BFP to 32)</td>
<td>CEFBR</td>
<td>RRE C SP Db Xi Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B399</td>
<td>CONVERT TO FIXED (long BFP to 32)</td>
<td>CFBR</td>
<td>RRE C SP Db Xi Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B39A</td>
<td>CONVERT TO FIXED (ext. BFP to 32)</td>
<td>CFBR</td>
<td>RRE C SP Db Xi Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B3A4</td>
<td>CONVERT FROM FIXED (64 to short BFP)</td>
<td>CEGBR</td>
<td>RRE N SP Db Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B3A5</td>
<td>CONVERT FROM FIXED (64 to long BFP)</td>
<td>CDBG</td>
<td>RRE N SP Db Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B3A6</td>
<td>CONVERT FROM FIXED (64 to ext. BFP)</td>
<td>CGXX</td>
<td>RRE N SP Db Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B3A8</td>
<td>CONVERT TO FIXED (short BFP to 64)</td>
<td>CGE BR</td>
<td>RRE C N SP Db Xi Xx</td>
<td>19-26</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 8 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mne- Code</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3A9</td>
<td>CONV TO FIXED (long BFP to 64)</td>
<td>CGDBR</td>
<td>Da Xi Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B3AA</td>
<td>CONV TO FIXED (ext. BFP to 64)</td>
<td>CGXBR</td>
<td>Da Xi Xx</td>
<td>19-26</td>
</tr>
<tr>
<td>B3B4</td>
<td>CONV FROM FIXED (32 to short HFP)</td>
<td>CEF R</td>
<td>Da</td>
<td>18-11</td>
</tr>
<tr>
<td>B3B5</td>
<td>CONV FROM FIXED (32 to long HFP)</td>
<td>CDFR R</td>
<td>Da</td>
<td>18-11</td>
</tr>
<tr>
<td>B3B6</td>
<td>CONV FROM FIXED (32 to ext. HFP)</td>
<td>CXFR R</td>
<td>Da</td>
<td>18-11</td>
</tr>
<tr>
<td>B3B8</td>
<td>CONV TO FIXED (short HFP to 32)</td>
<td>CFE R</td>
<td>Da</td>
<td>18-11</td>
</tr>
<tr>
<td>B3B9</td>
<td>CONV TO FIXED (long HFP to 32)</td>
<td>CDF R</td>
<td>Da</td>
<td>18-11</td>
</tr>
<tr>
<td>B3BA</td>
<td>CONV TO FIXED (ext. HFP to 32)</td>
<td>CXF R</td>
<td>Da</td>
<td>18-11</td>
</tr>
<tr>
<td>B3C4</td>
<td>CONV FROM FIXED (64 to short HFP)</td>
<td>CER R</td>
<td>SP</td>
<td>18-11</td>
</tr>
<tr>
<td>B3C5</td>
<td>CONV FROM FIXED (64 to long HFP)</td>
<td>CDR R</td>
<td>SP</td>
<td>18-11</td>
</tr>
<tr>
<td>B3C6</td>
<td>CONV FROM FIXED (64 to ext. HFP)</td>
<td>CXR R</td>
<td>SP</td>
<td>18-11</td>
</tr>
<tr>
<td>B3C8</td>
<td>CONV TO FIXED (short HFP to 64)</td>
<td>CFX R</td>
<td>SP</td>
<td>18-11</td>
</tr>
<tr>
<td>B3C9</td>
<td>CONV TO FIXED (long HFP to 64)</td>
<td>CDF R</td>
<td>SP</td>
<td>18-11</td>
</tr>
<tr>
<td>B3CA</td>
<td>CONV TO FIXED (ext. HFP to 64)</td>
<td>CXF R</td>
<td>SP</td>
<td>18-11</td>
</tr>
<tr>
<td>B6</td>
<td>STORE CONTROL (32)</td>
<td>STCTL RS</td>
<td>P A SP</td>
<td>10-107</td>
</tr>
<tr>
<td></td>
<td>B7 LOAD CONTROL (32)</td>
<td>LCTL RS RP A SP</td>
<td>IF</td>
<td>7-123</td>
</tr>
<tr>
<td>B900</td>
<td>LOAD POSITIVE (64)</td>
<td>LPGR R N RP A SP</td>
<td>IF</td>
<td>10-123</td>
</tr>
<tr>
<td>B901</td>
<td>LOAD NEGATIVE (64)</td>
<td>LNRG R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B902</td>
<td>LOAD AND TEST (64)</td>
<td>LTRG R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B903</td>
<td>LOAD COMPLEMENT (64)</td>
<td>LGCR R C N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B904</td>
<td>LOAD (64)</td>
<td>LGR R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B905</td>
<td>LOAD USING REAL ADDRESS (64)</td>
<td>LURAG R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B908</td>
<td>ADD (64)</td>
<td>SGR R C N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B909</td>
<td>SUBTRACT (64)</td>
<td>ALGR R C N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B90B</td>
<td>SUBTRACT LOGICAL (64)</td>
<td>SLGR R C N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B90C</td>
<td>MULTIPLY SINGLE (64)</td>
<td>MSGR R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B90D</td>
<td>DIVIDE SINGLE (64)</td>
<td>DSRG R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B90E</td>
<td>EXTRACT STACKED REGISTERS (64)</td>
<td>EREGG R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B90F</td>
<td>LOAD REVERSED (64)</td>
<td>LVRG R N</td>
<td>IF</td>
<td>7-126</td>
</tr>
<tr>
<td>B910</td>
<td>LOAD POSITIVE (64&lt;32)</td>
<td>LPGR R C N</td>
<td>IF</td>
<td>7-131</td>
</tr>
<tr>
<td>B911</td>
<td>LOAD NEGATIVE (64&lt;32)</td>
<td>LNRG R C N</td>
<td>IF</td>
<td>7-131</td>
</tr>
<tr>
<td>B912</td>
<td>LOAD AND TEST (64&lt;32)</td>
<td>LTRG R C N</td>
<td>IF</td>
<td>7-131</td>
</tr>
<tr>
<td>B913</td>
<td>LOAD COMPLEMENT (64&lt;32)</td>
<td>LGCR R C N</td>
<td>IF</td>
<td>7-131</td>
</tr>
<tr>
<td>B914</td>
<td>LOAD (64&lt;32)</td>
<td>LGF R N</td>
<td>IF</td>
<td>7-131</td>
</tr>
<tr>
<td>B916</td>
<td>LOAD LOGICAL (64&lt;32)</td>
<td>LLGR R N</td>
<td>IF</td>
<td>7-129</td>
</tr>
<tr>
<td>B917</td>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>LLGTR R N</td>
<td>IF</td>
<td>7-18</td>
</tr>
<tr>
<td>B918</td>
<td>ADD (64&lt;32)</td>
<td>AGF R C N</td>
<td>IF</td>
<td>7-18</td>
</tr>
<tr>
<td>B919</td>
<td>SUBTRACT (64&lt;32)</td>
<td>SGRF R C N</td>
<td>IF</td>
<td>7-18</td>
</tr>
<tr>
<td>B91A</td>
<td>ADD LOGICAL (64&lt;32)</td>
<td>ALGRF R C N</td>
<td>IF</td>
<td>7-18</td>
</tr>
<tr>
<td>B91B</td>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>SLGR R C N</td>
<td>SP</td>
<td>7-128</td>
</tr>
<tr>
<td>B91C</td>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>MSGF R N</td>
<td>IK</td>
<td>7-151</td>
</tr>
<tr>
<td>B91D</td>
<td>DIVIDE SINGLE (64&lt;32)</td>
<td>DSGF R N</td>
<td>IK</td>
<td>7-117</td>
</tr>
<tr>
<td>B91E</td>
<td>COMPUTE MESSAGE AUTHENTICATION CODE</td>
<td>KMAC R C M S A SP</td>
<td>GM I1</td>
<td>7-91</td>
</tr>
<tr>
<td>B91F</td>
<td>LOAD REVERSED (32)</td>
<td>LRVR R N3</td>
<td>SP</td>
<td>7-132</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 9 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>B920</td>
<td>COMPARE (64)</td>
<td>CGR</td>
<td>RRE C N</td>
<td>7-45</td>
</tr>
<tr>
<td>B921</td>
<td>COMPARE LOGICAL (64)</td>
<td>CLGR</td>
<td>RRE C N</td>
<td>7-66</td>
</tr>
<tr>
<td>B925</td>
<td>STORE USING REAL ADDRESS (64)</td>
<td>STURG</td>
<td>RRE N P A1 SP</td>
<td>10-124</td>
</tr>
<tr>
<td>B92E</td>
<td>CIPHER MESSAGE</td>
<td>KM</td>
<td>RRE C MS A SP</td>
<td>GM I1 ST</td>
</tr>
<tr>
<td>B92F</td>
<td>CIPHER MESSAGE WITH CHAINING</td>
<td>KMC</td>
<td>RRE C MS A SP</td>
<td>GM I1 ST</td>
</tr>
<tr>
<td>B930</td>
<td>COMPARE (64&lt;32)</td>
<td>CGFR</td>
<td>RRE C N</td>
<td>7-45</td>
</tr>
<tr>
<td>B931</td>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>CLGFR</td>
<td>RRE C N</td>
<td>7-66</td>
</tr>
<tr>
<td>B93E</td>
<td>COMPUTE INTERMEDIATE MESSAGE DIGEST</td>
<td>KIMD</td>
<td>RRE C MS A SP</td>
<td>GM I1 ST</td>
</tr>
<tr>
<td>B93F</td>
<td>COMPUTE LAST MESSAGE DIGEST</td>
<td>KLM</td>
<td>RRE C MS A SP</td>
<td>GM I1 ST</td>
</tr>
<tr>
<td>B946</td>
<td>BRANCH ON COUNT (64)</td>
<td>BCTGR</td>
<td>RRE N</td>
<td>7-26</td>
</tr>
<tr>
<td>B98/zerodot</td>
<td>AND (64)</td>
<td>NGR</td>
<td>RRE C N</td>
<td>7-2/zerodot</td>
</tr>
<tr>
<td>B981</td>
<td>OR (64)</td>
<td>OGR</td>
<td>RRE C N</td>
<td>7-151</td>
</tr>
<tr>
<td>B982</td>
<td>EXCLUSIVE OR (64)</td>
<td>XGR</td>
<td>RRE C N</td>
<td>7-118</td>
</tr>
<tr>
<td>B986</td>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>MLGR</td>
<td>RRE N</td>
<td>SP</td>
</tr>
<tr>
<td>B987</td>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>DLGR</td>
<td>RRE N</td>
<td>SP</td>
</tr>
<tr>
<td>B988</td>
<td>ADD LOGICAL WITH CARRY (64)</td>
<td>ALCGR</td>
<td>RRE C N</td>
<td>7-20</td>
</tr>
<tr>
<td>B989</td>
<td>SUBTRACT LOGICAL WITH BORROW (64)</td>
<td>SLBGR</td>
<td>RRE C N</td>
<td>7-198</td>
</tr>
<tr>
<td>B98A</td>
<td>ADD LOGICAL WITH CARRY (32)</td>
<td>RLCGR</td>
<td>RRE N3</td>
<td>7-15/zerodot</td>
</tr>
<tr>
<td>B98B</td>
<td>SUBTRACT LOGICAL WITH BORROW (32)</td>
<td>SLBGR</td>
<td>RRE N3</td>
<td>7-198</td>
</tr>
<tr>
<td>B98D</td>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>MLR</td>
<td>RRE N</td>
<td>7-15</td>
</tr>
<tr>
<td>B990</td>
<td>TRANSLATE TWO TO TWO</td>
<td>TRT</td>
<td>RRE C E2 A SP</td>
<td>GM</td>
</tr>
<tr>
<td>B991</td>
<td>TRANSLATE TWO TO ONE</td>
<td>TRT</td>
<td>RRE C E2 A SP</td>
<td>GM</td>
</tr>
<tr>
<td>B992</td>
<td>TRANSLATE ONE TO TWO</td>
<td>TROT</td>
<td>RRE C E2 A SP</td>
<td>GM</td>
</tr>
<tr>
<td>B993</td>
<td>TRANSLATE ONE TO ONE</td>
<td>TROT</td>
<td>RRE C E2 A SP</td>
<td>GM</td>
</tr>
<tr>
<td>B996</td>
<td>DIVIDE LOGICAL (32&lt;64)</td>
<td>DLR</td>
<td>RRE N3</td>
<td>SP</td>
</tr>
<tr>
<td>B99E</td>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>PTI</td>
<td>RRE RA Q A1 SP</td>
<td>Z E T e</td>
</tr>
<tr>
<td>B99F</td>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>SSAIR</td>
<td>RRE RA A1</td>
<td>Z E T e</td>
</tr>
<tr>
<td>B9A</td>
<td>EXTRACT PRIMARY ASN AND INSTANCE</td>
<td>EPAIR</td>
<td>RRE RA Q</td>
<td>SO</td>
</tr>
<tr>
<td>B9B</td>
<td>EXTRACT SECONDARY ASN AND INSTANCE</td>
<td>ESAIR</td>
<td>RRE RA Q</td>
<td>SO</td>
</tr>
<tr>
<td>B9D</td>
<td>EXTRACT AND SET EXTENDED AUTHORITY ASN</td>
<td>ESEA</td>
<td>RRE N P</td>
<td>10-21</td>
</tr>
<tr>
<td>B9E</td>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>PTAIR</td>
<td>RRE RA Q</td>
<td>A1 SP</td>
</tr>
<tr>
<td>B9F</td>
<td>PROGRAM TRANSFER WITH INSTANCE</td>
<td>SSAIR</td>
<td>RRE RA A1</td>
<td>Z E T e</td>
</tr>
<tr>
<td>B980</td>
<td>CONVERT UTF-8 TO UTF-32</td>
<td>CU14</td>
<td>RRE C E3 A SP</td>
<td>7-113</td>
</tr>
<tr>
<td>B981</td>
<td>CONVERT UTF-16 TO UTF-32</td>
<td>CU24</td>
<td>RRE C E3 A SP</td>
<td>7-98</td>
</tr>
<tr>
<td>B982</td>
<td>CONVERT UTF-32 TO UTF-8</td>
<td>CU14</td>
<td>RRE C E3 A SP</td>
<td>7-107</td>
</tr>
<tr>
<td>B983</td>
<td>CONVERT UTF-32 TO UTF-16</td>
<td>CU24</td>
<td>RRE C E3 A SP</td>
<td>7-104</td>
</tr>
<tr>
<td>B98E</td>
<td>SEARCH STRING UNICODE</td>
<td>CSR</td>
<td>RRE C E3 A SP</td>
<td>7-173</td>
</tr>
<tr>
<td>BA</td>
<td>COMPARE AND SWAP (32)</td>
<td>CS</td>
<td>RRE C A SP</td>
<td>7-53</td>
</tr>
<tr>
<td>BB</td>
<td>COMPARE DOUBLE AND SWAP (32)</td>
<td>CDS</td>
<td>RRE C A SP</td>
<td>7-53</td>
</tr>
<tr>
<td>BD</td>
<td>COMPARE LOGICAL C. UNDER MASK (low)</td>
<td>CLM</td>
<td>RRE C A</td>
<td>7-57</td>
</tr>
<tr>
<td>BE</td>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>SCM</td>
<td>RRE C A</td>
<td>7-181</td>
</tr>
<tr>
<td>BF</td>
<td>INSERT CHARACTERS UNDER MASK (low)</td>
<td>ICM</td>
<td>RRE C A</td>
<td>7-121</td>
</tr>
<tr>
<td>C00</td>
<td>LOAD ADDRESS RELATIVE LONG</td>
<td>LAR</td>
<td>RIL N3</td>
<td>7-125</td>
</tr>
<tr>
<td>C04</td>
<td>BRANCH RELATIVE ON CONDITION LONG</td>
<td>BRCL</td>
<td>RIL N3</td>
<td>7-29</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 10 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C05</td>
<td>BRANCH RELATIVE AND SAVE LONG</td>
<td>BRASL</td>
<td>RIL N3</td>
<td>GM</td>
</tr>
<tr>
<td>D0</td>
<td>TRANSLATE AND TEST REVERSED</td>
<td>TRTR</td>
<td>SS C E3</td>
<td>A</td>
</tr>
<tr>
<td>D1</td>
<td>MOVE NUMERICs</td>
<td>MCV</td>
<td>SS</td>
<td>A</td>
</tr>
<tr>
<td>D2</td>
<td>MOVE (character)</td>
<td>MVC</td>
<td>SS</td>
<td>A</td>
</tr>
<tr>
<td>D3</td>
<td>MOVE ZONES</td>
<td>MVZ</td>
<td>SS</td>
<td>A</td>
</tr>
<tr>
<td>D4</td>
<td>AND (character)</td>
<td>NC</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>D5</td>
<td>COMPARE LOGICAL (character)</td>
<td>CLC</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>D6</td>
<td>OR (character)</td>
<td>OC</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>D7</td>
<td>EXCLUSIVE OR (character)</td>
<td>XC</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>D9</td>
<td>MOVE WITH KEY</td>
<td>MVCK</td>
<td>SS C</td>
<td>Q A</td>
</tr>
<tr>
<td>D10</td>
<td>MOVE TO PRIMARY</td>
<td>MVCP</td>
<td>SS C</td>
<td>Q A</td>
</tr>
<tr>
<td>D11</td>
<td>MOVE TO SECONDARY</td>
<td>MVCS</td>
<td>SS C</td>
<td>Q A</td>
</tr>
<tr>
<td>D12</td>
<td>TRANSLATE</td>
<td>TR</td>
<td>SS</td>
<td>A</td>
</tr>
<tr>
<td>D13</td>
<td>TRANSLATE AND TEST</td>
<td>TRT</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>D14</td>
<td>EDIT</td>
<td>ED</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>D15</td>
<td>EDIT AND MARK</td>
<td>EDMK</td>
<td>SS C</td>
<td>A</td>
</tr>
<tr>
<td>E1</td>
<td>PACK UNICODE</td>
<td>PKU</td>
<td>SS E2</td>
<td>A SP</td>
</tr>
<tr>
<td>E2</td>
<td>UNPACK UNICODE</td>
<td>UNPKU</td>
<td>SS C E2</td>
<td>A SP</td>
</tr>
<tr>
<td>E303</td>
<td>LOAD REAL ADDRESS (64)</td>
<td>LRAG</td>
<td>RXY C N</td>
<td>P A1</td>
</tr>
<tr>
<td>E304</td>
<td>LOAD (64)</td>
<td>LG</td>
<td>RXY</td>
<td>N A</td>
</tr>
<tr>
<td>E306</td>
<td>CONVERT TO BINARY (32)</td>
<td>CVBY</td>
<td>RXY LD</td>
<td>A</td>
</tr>
<tr>
<td>E306</td>
<td>ADD (64)</td>
<td>AG</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E307</td>
<td>SUBTRACT (64)</td>
<td>SG</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E30A</td>
<td>ADD LOGICAL (64)</td>
<td>ALG</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E30B</td>
<td>SUBTRACT LOGICAL (64)</td>
<td>SLG</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E30C</td>
<td>MULTIPLY SINGLE (64)</td>
<td>MSG</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E30D</td>
<td>DIVIDE SINGLE (64)</td>
<td>DSG</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E30E</td>
<td>CONVERT TO BINARY (64)</td>
<td>CVBG</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E30F</td>
<td>LOAD (64&lt;32)</td>
<td>LRG</td>
<td>RXY N A</td>
<td>A</td>
</tr>
<tr>
<td>E313</td>
<td>LOAD REAL ADDRESS (32)</td>
<td>LRAY</td>
<td>RXY C LD</td>
<td>P A1</td>
</tr>
<tr>
<td>E314</td>
<td>LOAD (64&lt;32)</td>
<td>LGF</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E315</td>
<td>LOAD HALFWORD (64)</td>
<td>LGH</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E316</td>
<td>LOAD LOGICAL (64&lt;32)</td>
<td>LLGF</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E317</td>
<td>LOAD LOGICAL THIRTY ONE BITS</td>
<td>LLGT</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E318</td>
<td>ADD (64&lt;32)</td>
<td>AGF</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E319</td>
<td>SUBTRACT (64&lt;32)</td>
<td>SGF</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E31A</td>
<td>ADD LOGICAL (64&lt;32)</td>
<td>ALGF</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E31B</td>
<td>SUBTRACT LOGICAL (64&lt;32)</td>
<td>SLGF</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E31C</td>
<td>MULTIPLY SINGLE (64&lt;32)</td>
<td>MSGF</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E31D</td>
<td>DIVIDE SINGLE (64&lt;32)</td>
<td>DSGF</td>
<td>RXY N</td>
<td>A</td>
</tr>
<tr>
<td>E31E</td>
<td>LOAD REVERSED (32)</td>
<td>LRV</td>
<td>RXY N3</td>
<td>A</td>
</tr>
<tr>
<td>E31F</td>
<td>LOAD REVERSED (16)</td>
<td>LRVH</td>
<td>RXY N3</td>
<td>A</td>
</tr>
<tr>
<td>E320</td>
<td>COMPARE (64)</td>
<td>CG</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E321</td>
<td>COMPARE LOGICAL (64)</td>
<td>CLG</td>
<td>RXY C N</td>
<td>A</td>
</tr>
<tr>
<td>E324</td>
<td>STORE (64)</td>
<td>STG</td>
<td>RXY N</td>
<td>A</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 11 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>E326</td>
<td>CONVERT TO DECIMAL (32)</td>
<td>CVDY RXY LD A</td>
<td>ST</td>
<td>B2 7-98</td>
</tr>
<tr>
<td>E32E</td>
<td>CONVERT TO DECIMAL (64)</td>
<td>CVDG RXY N A</td>
<td>ST</td>
<td>B2 7-98</td>
</tr>
<tr>
<td>E32F</td>
<td>STORE REVERSED (64)</td>
<td>STRVG RXY N A</td>
<td>ST</td>
<td>B2 7-187</td>
</tr>
<tr>
<td>E330</td>
<td>COMPARE (64&lt;32)</td>
<td>CGF RXY C N A</td>
<td>ST</td>
<td>B2 7-46</td>
</tr>
<tr>
<td>E331</td>
<td>COMPARE LOGICAL (64&lt;32)</td>
<td>CLGF RXY C N A</td>
<td>ST</td>
<td>B2 7-56</td>
</tr>
<tr>
<td>E33E</td>
<td>STORE REVERSED (32)</td>
<td>STRV RXY N3 A</td>
<td>ST</td>
<td>B2 7-187</td>
</tr>
<tr>
<td>E33F</td>
<td>STORE REVERSED (16)</td>
<td>STRVH RXY N3 A</td>
<td>ST</td>
<td>B2 7-187</td>
</tr>
<tr>
<td>E346</td>
<td>BRANCH ON COUNT (64)</td>
<td>BCTG RXY N</td>
<td>ST</td>
<td>B2 7-26</td>
</tr>
<tr>
<td>E35/zerodot</td>
<td>STORE (32)</td>
<td>STY RXY LD A</td>
<td>ST</td>
<td>B2 7-18/zerodot</td>
</tr>
<tr>
<td>E351</td>
<td>MULTIPLY SINGLE (32)</td>
<td>LBY RXY LD A</td>
<td>ST</td>
<td>B2 7-123</td>
</tr>
<tr>
<td>E354</td>
<td>AND (32)</td>
<td>NY RXY C LD A</td>
<td>ST</td>
<td>B2 7-20</td>
</tr>
<tr>
<td>E355</td>
<td>OR (32)</td>
<td>OG RXY C LD A</td>
<td>ST</td>
<td>B2 7-56</td>
</tr>
<tr>
<td>E356</td>
<td>EXCLUSIVE OR (32)</td>
<td>XG RXY C LD A</td>
<td>ST</td>
<td>B2 7-152</td>
</tr>
<tr>
<td>E357</td>
<td>COMPARE LOGICAL (32)</td>
<td>CY RXY C LD A</td>
<td>ST</td>
<td>B2 7-46</td>
</tr>
<tr>
<td>E35A</td>
<td>ADD (32)</td>
<td>AY RXY C LD A</td>
<td>IF</td>
<td>B2 7-18</td>
</tr>
<tr>
<td>E35B</td>
<td>SUBTRACT (32)</td>
<td>SY RXY C LD A</td>
<td>IF</td>
<td>B2 7-187</td>
</tr>
<tr>
<td>E35E</td>
<td>ADD LOGICAL (32)</td>
<td>ALY RXY C LD A</td>
<td>IF</td>
<td>B2 7-19</td>
</tr>
<tr>
<td>E35F</td>
<td>SUBTRACT LOGICAL (32)</td>
<td>SLY RXY C LD A</td>
<td>IF</td>
<td>B2 7-19</td>
</tr>
<tr>
<td>E359</td>
<td>COMPARE (32)</td>
<td>CY RXY C LD A</td>
<td>ST</td>
<td>B2 7-46</td>
</tr>
<tr>
<td>E370</td>
<td>STORE HALFWORD</td>
<td>STHY RXY LD A</td>
<td>ST</td>
<td>B2 7-185</td>
</tr>
<tr>
<td>E371</td>
<td>LOAD ADDRESS</td>
<td>LAY RXY LD A</td>
<td>ST</td>
<td>B2 7-124</td>
</tr>
<tr>
<td>E372</td>
<td>STORE CHARACTER</td>
<td>STCY RXY LD A</td>
<td>ST</td>
<td>B2 7-181</td>
</tr>
<tr>
<td>E373</td>
<td>INSERT CHARACTER</td>
<td>ICY RXY LD A</td>
<td>ST</td>
<td>B2 7-121</td>
</tr>
<tr>
<td>E374</td>
<td>LOAD BYTE (32)</td>
<td>LB RXY LD A</td>
<td>ST</td>
<td>B2 7-126</td>
</tr>
<tr>
<td>E377</td>
<td>LOAD BYTE (64)</td>
<td>LGB RXY LD A</td>
<td>ST</td>
<td>B2 7-126</td>
</tr>
<tr>
<td>E378</td>
<td>LOAD HALFWORD (32)</td>
<td>LHY RXY LD A</td>
<td>ST</td>
<td>B2 7-127</td>
</tr>
<tr>
<td>E379</td>
<td>COMPARE HALFWORD</td>
<td>CHY RXY C LD A</td>
<td>ST</td>
<td>B2 7-55</td>
</tr>
<tr>
<td>E37A</td>
<td>ADD HALFWORD</td>
<td>AHY RXY C LD A</td>
<td>ST</td>
<td>B2 7-18</td>
</tr>
<tr>
<td>E37B</td>
<td>SUBTRACT HALFWORD</td>
<td>SHY RXY C LD A</td>
<td>ST</td>
<td>B2 7-188</td>
</tr>
<tr>
<td>E38/zerodot</td>
<td>AND (64)</td>
<td>NG RXY C N A</td>
<td>ST</td>
<td>B2 7-20</td>
</tr>
<tr>
<td>E381</td>
<td>OR (64)</td>
<td>OG RXY C N A</td>
<td>ST</td>
<td>B2 7-152</td>
</tr>
<tr>
<td>E382</td>
<td>EXCLUSIVE OR (64)</td>
<td>XG RXY C N A</td>
<td>ST</td>
<td>B2 7-118</td>
</tr>
<tr>
<td>E386</td>
<td>MULTIPLY LOGICAL (128&lt;64)</td>
<td>MLG RXY N A SP</td>
<td>ST</td>
<td>B2 7-150</td>
</tr>
<tr>
<td>E387</td>
<td>DIVIDE LOGICAL (64&lt;128)</td>
<td>DLG RXY N A SP</td>
<td>ST</td>
<td>B2 7-117</td>
</tr>
<tr>
<td>E388</td>
<td>ADD LOGICAL WITH CARRY (64)</td>
<td>ALCG RXY C N A</td>
<td>ST</td>
<td>B2 7-20</td>
</tr>
<tr>
<td>E38E</td>
<td>STORE PAIR TO QUADWORD</td>
<td>STPQ RXY N A SP</td>
<td>ST</td>
<td>B2 7-186</td>
</tr>
<tr>
<td>E38F</td>
<td>LOAD PAIR FROM QUADWORD</td>
<td>LPQ RXY N A SP</td>
<td>ST</td>
<td>B2 7-131</td>
</tr>
<tr>
<td>E390</td>
<td>LOAD LOGICAL CHARACTER</td>
<td>LLGC RXY N A</td>
<td>ST</td>
<td>B2 7-128</td>
</tr>
<tr>
<td>E391</td>
<td>LOAD LOGICAL HALFWORD</td>
<td>LLGH RXY N A</td>
<td>ST</td>
<td>B2 7-128</td>
</tr>
<tr>
<td>E396</td>
<td>MULTIPLY LOGICAL (64&lt;32)</td>
<td>ML RXY N3 A SP</td>
<td>ST</td>
<td>B2 7-150</td>
</tr>
<tr>
<td>E399</td>
<td>SUBTRACT LOGICAL WITH BORROW (32)</td>
<td>SLC RXY C N A</td>
<td>ST</td>
<td>B2 7-189</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 12 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mne- monic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>E500</td>
<td>LOAD ADDRESS SPACE PARAMETERS</td>
<td>LASP</td>
<td>P A SP</td>
<td>B1</td>
</tr>
<tr>
<td>E501</td>
<td>TEST PROTECTION</td>
<td>TPRQ</td>
<td>P A SP</td>
<td>B1</td>
</tr>
<tr>
<td>E502</td>
<td>STORE REAL ADDRESS</td>
<td>STRAG</td>
<td>P A SP</td>
<td>B1 BP</td>
</tr>
<tr>
<td>E50E</td>
<td>MOVE WITH SOURCE KEY</td>
<td>MVCSDK</td>
<td>Q A GM</td>
<td>B1 B2</td>
</tr>
<tr>
<td>E50F</td>
<td>MOVE WITH DESTINATION KEY</td>
<td>MVCSDK</td>
<td>Q A GM</td>
<td>B1 B2</td>
</tr>
<tr>
<td>E8</td>
<td>MOVE INVERSE</td>
<td>MVCIN</td>
<td>A SP</td>
<td>ST</td>
</tr>
<tr>
<td>E9</td>
<td>PACK ASCII</td>
<td>PKA</td>
<td>A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EA</td>
<td>UNPACK ASCII</td>
<td>UNPKA</td>
<td>A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB04</td>
<td>LOAD MULTIPLE (64)</td>
<td>LMG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB0A</td>
<td>SHIFT RIGHT SINGLE (64)</td>
<td>SRAG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB0B</td>
<td>SHIFT LEFT SINGLE (64)</td>
<td>SLAG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB0C</td>
<td>SHIFT RIGHT SINGLE LOGICAL (64)</td>
<td>SRLG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB0D</td>
<td>SHIFT LEFT SINGLE LOGICAL (64)</td>
<td>SLLG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB0F</td>
<td>TRACE (64)</td>
<td>TRACG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB14</td>
<td>COMPARE AND SWAP (32)</td>
<td>CSY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB1C</td>
<td>ROTATE LEFT SINGLE LOGICAL (64)</td>
<td>RLLG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB1D</td>
<td>ROTATE LEFT SINGLE LOGICAL (32)</td>
<td>RLL</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB20</td>
<td>COMPARE LOGICAL C. UNDER MASK (high)</td>
<td>CLMH</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB21</td>
<td>COMPARE LOGICAL C. UNDER MASK (low)</td>
<td>CLMY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB24</td>
<td>STORE MULTIPLE (64)</td>
<td>STMG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB25</td>
<td>STORE CONTROL (64)</td>
<td>STCTG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB26</td>
<td>STORE MULTIPLE HIGH</td>
<td>STMH</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB2C</td>
<td>STORE CHARACTERS UNDER MASK (high)</td>
<td>STCMH</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB2D</td>
<td>STORE CHARACTERS UNDER MASK (low)</td>
<td>STCMY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB2F</td>
<td>LOAD CONTROL (64)</td>
<td>LCTLG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB30</td>
<td>COMPARE AND SWAP (64)</td>
<td>CSG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB31</td>
<td>COMPARE DOUBLE AND SWAP (32)</td>
<td>CDSY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB3E</td>
<td>COMPARE DOUBLE AND SWAP (64)</td>
<td>CDSG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB44</td>
<td>BRANCH ON INDEX HIGH (64)</td>
<td>BXHG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB45</td>
<td>BRANCH ON INDEX LOW OR EQUAL (64)</td>
<td>BXLEG</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB51</td>
<td>TEST UNDER MASK</td>
<td>TMY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB52</td>
<td>MOVE (immediate)</td>
<td>MVX</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB54</td>
<td>AND (immediate)</td>
<td>NIX</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB55</td>
<td>COMPARE LOGICAL (immediate)</td>
<td>CLIX</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB56</td>
<td>OR (immediate)</td>
<td>OIX</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB57</td>
<td>EXCLUSIVE OR (immediate)</td>
<td>XIY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB80</td>
<td>INSERT CHARACTERS UNDER MASK (high)</td>
<td>ICMH</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB81</td>
<td>INSERT CHARACTERS UNDER MASK (low)</td>
<td>ICML</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB85</td>
<td>MOVE LONG UNICODE</td>
<td>MVCLU</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB8F</td>
<td>COMPARE LOGICAL LONG UNICODE</td>
<td>CLCLU</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB90</td>
<td>STORE MULTIPLE (32)</td>
<td>STMY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB96</td>
<td>LOAD MULTIPLE HIGH</td>
<td>LMH</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB98</td>
<td>LOAD MULTIPLE (32)</td>
<td>LMY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB9A</td>
<td>LOAD ACCESS MULTIPLE</td>
<td>LAMY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
<tr>
<td>EB9B</td>
<td>STORE ACCESS MULTIPLE</td>
<td>STMAY</td>
<td>P A SP</td>
<td>ST</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 13 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mne-</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBC0</td>
<td>TEST DECIMAL</td>
<td>TP</td>
<td>RSL C E2</td>
<td>A</td>
</tr>
<tr>
<td>EC44</td>
<td>BRANCH RELATIVE ON INDEX HIGH (64)</td>
<td>BRXHG</td>
<td>RIE N</td>
<td>Da</td>
</tr>
<tr>
<td>EC45</td>
<td>BRANCH RELATIVE ON INDEX L OR E (64)</td>
<td>BRXLG</td>
<td>RIE N</td>
<td>Da</td>
</tr>
<tr>
<td>ED04</td>
<td>LOAD LENGTHENED (short to long BFP)</td>
<td>LDEB</td>
<td>RXE</td>
<td>A</td>
</tr>
<tr>
<td>ED05</td>
<td>LOAD LENGTHENED (long to ext. BFP)</td>
<td>LXDB</td>
<td>RXE</td>
<td>A SP</td>
</tr>
<tr>
<td>ED06</td>
<td>LOAD LENGTHENED (short to ext. BFP)</td>
<td>LXEB</td>
<td>RXE</td>
<td>A SP</td>
</tr>
<tr>
<td>ED07</td>
<td>MULTIPLY (long to extended BFP)</td>
<td>MDEB</td>
<td>RXE</td>
<td>A Db Xi</td>
</tr>
<tr>
<td>ED08</td>
<td>COMPARE AND SIGNAL (short BFP)</td>
<td>KEB</td>
<td>RXE C</td>
<td>A Db Xi</td>
</tr>
<tr>
<td>ED09</td>
<td>COMPARE (short BFP)</td>
<td>CEB</td>
<td>RXE C</td>
<td>A Db Xi</td>
</tr>
<tr>
<td>ED0A</td>
<td>ADD (short BFP)</td>
<td>AEB</td>
<td>RXE C</td>
<td>A Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ED0B</td>
<td>SUBTRACT (short BFP)</td>
<td>SEB</td>
<td>RXE C</td>
<td>A Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ED0C</td>
<td>MULTIPLY (short to long BFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0D</td>
<td>DIVIDE (short BFP)</td>
<td>DMB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0E</td>
<td>MULTIPLY AND ADD (short BFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0F</td>
<td>MULTIPLY AND SUBTRACT (short BFP)</td>
<td>MSEB</td>
<td>RXF A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0G</td>
<td>SUBTRACT (long BFP)</td>
<td>SDB</td>
<td>RXE C</td>
<td>A Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ED0H</td>
<td>SQUARE ROOT (short BFP)</td>
<td>SQEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0I</td>
<td>SQUARE ROOT (long BFP)</td>
<td>SQCB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0J</td>
<td>MULTIPLY (long BFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0K</td>
<td>DIVIDE (long BFP)</td>
<td>DDB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0L</td>
<td>MULTIPLY AND ADD (long BFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0M</td>
<td>MULTIPLY AND SUBTRACT (long BFP)</td>
<td>MSEB</td>
<td>RXF A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0N</td>
<td>SUBTRACT (long BFP)</td>
<td>SDB</td>
<td>RXE C</td>
<td>A Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ED0O</td>
<td>SQUARE ROOT (short HFP)</td>
<td>SQE</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0P</td>
<td>SQUARE ROOT (long HFP)</td>
<td>SQD</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0Q</td>
<td>MULTIPLY (long HFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0R</td>
<td>DIVIDE (long HFP)</td>
<td>DDB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0S</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0T</td>
<td>MULTIPLY AND SUBTRACT (long HFP)</td>
<td>MSEB</td>
<td>RXF A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0U</td>
<td>SUBTRACT (long HFP)</td>
<td>SDB</td>
<td>RXE C</td>
<td>A Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ED0V</td>
<td>SQUARE ROOT (short HFP)</td>
<td>SQE</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0W</td>
<td>SQUARE ROOT (long HFP)</td>
<td>SQD</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0X</td>
<td>MULTIPLY (long HFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0Y</td>
<td>DIVIDE (long HFP)</td>
<td>DDB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED0Z</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED1A</td>
<td>ADD (long HFP)</td>
<td>ADD</td>
<td>RXE C A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED1B</td>
<td>SUBTRACT (long HFP)</td>
<td>SDB</td>
<td>RXE C</td>
<td>A Db Xi Xo Xu Xx</td>
</tr>
<tr>
<td>ED1C</td>
<td>MULTIPLY (long HFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED1D</td>
<td>DIVIDE (long HFP)</td>
<td>DDB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED1E</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>MDEB</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED1F</td>
<td>MULTIPLY AND SUBTRACT (long HFP)</td>
<td>MSEB</td>
<td>RXF A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED24</td>
<td>LOAD LENGTHENED (short to long HFP)</td>
<td>LDE</td>
<td>RXE A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED25</td>
<td>LOAD LENGTHENED (long to ext. HFP)</td>
<td>LXE</td>
<td>RXE A SP Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED26</td>
<td>LOAD LENGTHENED (short to ext. HFP)</td>
<td>LXE</td>
<td>RXE A SP Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED27</td>
<td>MULTIPLY AND ADD (short HFP)</td>
<td>MAEB</td>
<td>RXF HM A Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>ED2E</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>MDEB</td>
<td>RXF HM A Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>ED2F</td>
<td>MULTIPLY AND SUBTRACT (short HFP)</td>
<td>MSEB</td>
<td>RXF A Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>ED34</td>
<td>SQUARE ROOT (short HFP)</td>
<td>SQE</td>
<td>RXE A Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>ED35</td>
<td>SQUARE ROOT (long HFP)</td>
<td>SQD</td>
<td>RXE A Da SQ</td>
<td>B2</td>
</tr>
<tr>
<td>ED37</td>
<td>MULTIPLY (short HFP)</td>
<td>MDEB</td>
<td>RXE A Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>ED38</td>
<td>DIVIDE (short HFP)</td>
<td>DDEB</td>
<td>RXE A Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>ED39</td>
<td>MULTIPLY AND ADD (long HFP)</td>
<td>MDEB</td>
<td>RXF HM A Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>ED3F</td>
<td>MULTIPLY AND SUBTRACT (long HFP)</td>
<td>MSEB</td>
<td>RXF A Da EU EO</td>
<td>B2</td>
</tr>
<tr>
<td>ED40</td>
<td>LOAD (short)</td>
<td>LEY</td>
<td>RXY LD A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED41</td>
<td>LOAD (long)</td>
<td>LDY</td>
<td>RXY LD A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED42</td>
<td>STORE (short)</td>
<td>STEY</td>
<td>RXY LD A Da</td>
<td>B2</td>
</tr>
<tr>
<td>ED43</td>
<td>STORE (long)</td>
<td>STDY</td>
<td>RXY LD A Da</td>
<td>B2</td>
</tr>
<tr>
<td>EE</td>
<td>PERFORM LOCKED OPERATION</td>
<td>PLO</td>
<td>SS C A SP $ GM</td>
<td>B2</td>
</tr>
<tr>
<td>EF</td>
<td>LOAD MULTIPLE DISJOINT</td>
<td>LMD</td>
<td>SS N A</td>
<td>B2</td>
</tr>
</tbody>
</table>

Figure B-3 (Part 14 of 15). Instructions Arranged by Operation Code
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Name</th>
<th>Mnemonic</th>
<th>Characteristics</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>SHIFT AND ROUND DECIMAL</td>
<td>SRP</td>
<td>C</td>
<td>8-11</td>
</tr>
<tr>
<td>F1</td>
<td>MOVE WITH OFFSET</td>
<td>MVO</td>
<td>S</td>
<td>ST</td>
</tr>
<tr>
<td>F2</td>
<td>PACK</td>
<td>PACK</td>
<td>S</td>
<td>ST</td>
</tr>
<tr>
<td>F3</td>
<td>UNPACK</td>
<td>UNPK</td>
<td>S</td>
<td>ST</td>
</tr>
<tr>
<td>F8</td>
<td>ZERO AND ADD</td>
<td>ZAP</td>
<td>C</td>
<td>ST</td>
</tr>
<tr>
<td>F9</td>
<td>COMPARE DECIMAL</td>
<td>CP</td>
<td>C</td>
<td>ST</td>
</tr>
<tr>
<td>FA</td>
<td>ADD DECIMAL</td>
<td>AP</td>
<td>C</td>
<td>ST</td>
</tr>
<tr>
<td>FB</td>
<td>SUBTRACT DECIMAL</td>
<td>SP</td>
<td>C</td>
<td>ST</td>
</tr>
<tr>
<td>FC</td>
<td>MULTIPLY DECIMAL</td>
<td>MP</td>
<td>S</td>
<td>ST</td>
</tr>
<tr>
<td>FD</td>
<td>DIVIDE DECIMAL</td>
<td>DP</td>
<td>S</td>
<td>ST</td>
</tr>
</tbody>
</table>

*Figure B-3 (Part 15 of 15). Instructions Arranged by Operation Code*
Appendix C. Condition-Code Settings

This appendix lists the condition-code setting for instructions in z/Architecture which set the condition code. In addition to those instructions listed which set the condition code, the condition code may be changed by DIAGNOSE and the target of EXECUTE. The condition code is loaded by LOAD PSW, LOAD PSW EXTENDED, PROGRAM RETURN, RESUME PROGRAM, and SET PROGRAM MASK and by an interruption. The condition code is set to zero by initial CPU reset and is loaded by the successful conclusion of the initial-program-loading sequence.

Some models may offer instructions which set the condition code and do not appear in this document, such as those provided for assists or as part of special or custom features.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD (gen)</strong></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>Overflow</td>
</tr>
<tr>
<td><strong>ADD (BFP)</strong></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
</tr>
<tr>
<td><strong>ADD DECIMAL</strong></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>Overflow</td>
</tr>
<tr>
<td><strong>ADD HALFWORD</strong></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>Overflow</td>
</tr>
<tr>
<td><strong>ADD HALFWORD IMMEDIATE</strong></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>Overflow</td>
</tr>
<tr>
<td><strong>ADD LOGICAL</strong></td>
<td>Zero, no carry</td>
<td>Not zero, no carry</td>
<td>Zero, carry</td>
<td>Not zero, carry</td>
</tr>
<tr>
<td><strong>ADD LOGICAL WITH CARRY</strong></td>
<td>Zero, no carry</td>
<td>Not zero, no carry</td>
<td>Zero, carry</td>
<td>Not zero, carry</td>
</tr>
<tr>
<td><strong>ADD NORMALIZED</strong></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>--</td>
</tr>
<tr>
<td><strong>ADD UNNORMALIZED</strong></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>--</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>Zero</td>
<td>Not zero</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td><strong>CANCEL SUBCHANNEL</strong></td>
<td>Function initiated</td>
<td>--</td>
<td>--</td>
<td>Not operational</td>
</tr>
<tr>
<td><strong>CHECKSUM</strong></td>
<td>Checksum complete</td>
<td>--</td>
<td>--</td>
<td>CPU-determined completion</td>
</tr>
<tr>
<td><strong>CIPHER MESSAGE</strong></td>
<td>Normal completion</td>
<td>--</td>
<td>--</td>
<td>Partial completion</td>
</tr>
<tr>
<td><strong>CIPHER MESSAGE WITH CHAINING</strong></td>
<td>Normal completion</td>
<td>--</td>
<td>--</td>
<td>Partial completion</td>
</tr>
<tr>
<td><strong>CLEAR SUBCHANNEL</strong></td>
<td>Function initiated</td>
<td>--</td>
<td>--</td>
<td>Not operational</td>
</tr>
<tr>
<td><strong>COMPARE (gen, HFP)</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE (BFP)</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>Unordered</td>
</tr>
<tr>
<td><strong>COMPARE AND FORM CODEWORD</strong></td>
<td>Equal</td>
<td>OCB=0: low</td>
<td>OCB=0: high</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE AND SIGNAL</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>Unordered</td>
</tr>
<tr>
<td><strong>COMPARE AND SWAP</strong></td>
<td>Equal</td>
<td>Not equal</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE AND SWAP AND PURGE</strong></td>
<td>Equal</td>
<td>Not equal</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE DECIMAL</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE DOUBLE AND SWAP</strong></td>
<td>Equal</td>
<td>Not equal</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE HALFWORD</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE HALFWORD IMMEDIATE</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE LOGICAL</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE LOGICAL CHARACTERS</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>--</td>
</tr>
<tr>
<td><strong>UNDER MASK</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>--</td>
</tr>
<tr>
<td><strong>COMPARE LOGICAL LONG</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>CPU-determined completion</td>
</tr>
<tr>
<td><strong>COMPARE LOGICAL LONG EXTENDED</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>CPU-determined completion</td>
</tr>
<tr>
<td><strong>COMPARE LOGICAL LONG UNICODE</strong></td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>CPU-determined completion</td>
</tr>
</tbody>
</table>

*Figure C-1 (Part 1 of 6). Summary of Condition-Code Settings*
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td>COMPARE LOGICAL STRING</td>
<td>Equal</td>
</tr>
<tr>
<td>COMPARE UNTIL SUBSTRING EQUAL</td>
<td>Equal subrings Op2 processed</td>
</tr>
<tr>
<td>COMPRESSION CALL</td>
<td>--</td>
</tr>
<tr>
<td>COMPUTE INTERMEDIATE MESSAGE DIGEST</td>
<td>Normal completion</td>
</tr>
<tr>
<td>COMPUTE LAST MESSAGE DIGEST</td>
<td>Normal completion</td>
</tr>
<tr>
<td>COMPUTE MESSAGE AUTHENTICATION CODE</td>
<td>Normal completion</td>
</tr>
<tr>
<td>CONVERT BFP TO HFP</td>
<td>Zero</td>
</tr>
<tr>
<td>CONVERT HFP TO BFP</td>
<td>Zero</td>
</tr>
<tr>
<td>CONVERT TO FIXED</td>
<td>Zero</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-32</td>
<td>Data processed</td>
</tr>
<tr>
<td>CONVERT UTF-16 TO UTF-8</td>
<td>Data processed</td>
</tr>
<tr>
<td>CONVERT UTF-32 TO UTF-16</td>
<td>Data processed</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UTF-16</td>
<td>Data processed</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UTF-32</td>
<td>Data processed</td>
</tr>
<tr>
<td>DIVIDE TO INTEGER</td>
<td>Remainder complete; normal quotient</td>
</tr>
<tr>
<td>EDIT</td>
<td>Zero</td>
</tr>
<tr>
<td>EDIT AND MARK</td>
<td>Zero</td>
</tr>
<tr>
<td>EXCLUSIVE OR</td>
<td>Zero</td>
</tr>
<tr>
<td>EXTRACT STACKED STATE</td>
<td>Branch state entry</td>
</tr>
</tbody>
</table>

Figure C-1 (Part 2 of 6). Summary of Condition-Code Settings
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT SUBCHANNEL</td>
<td>Function initiated</td>
<td></td>
<td>Status-pending with other than intermediate status</td>
<td>Busy</td>
<td>Not operational</td>
</tr>
<tr>
<td>INSERT ADDRESS SPACE CONTROL</td>
<td>Primary-space mode initiated</td>
<td>All zeros</td>
<td>First bit one</td>
<td>Access-register mode</td>
<td>Home-space mode</td>
</tr>
<tr>
<td>INSERT CHARACTERS UNDER MASK</td>
<td>Parameters loaded</td>
<td>Operation pending</td>
<td>Secondary-space mode</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>LOAD ADDRESS SPACE PARAMETERS</td>
<td></td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD AND TEST (gen, HFP)</td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD AND TEST (BFP)</td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (gen)</td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (BFP)</td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD COMPLEMENT (HFP)</td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD NEGATIVE (BFP)</td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD POSITIVE (gen)</td>
<td>Zero</td>
<td>-- invalid</td>
<td>&gt; zero invalid</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD POSITIVE (BFP)</td>
<td>Zero available</td>
<td>-- invalid</td>
<td>&gt; zero invalid</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD POSITIVE (HFP)</td>
<td>Zero Translation</td>
<td>-- invalid</td>
<td>&gt; zero Translation</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>LOAD REAL ADDRESS</td>
<td>Translation</td>
<td>-- invalid</td>
<td>&gt; zero Translation</td>
<td>NaN</td>
<td>--</td>
</tr>
<tr>
<td>MODIFY SUBCHANNEL</td>
<td>SCHIB information placed in subchannel</td>
<td>Status-pending</td>
<td>Busy</td>
<td>Not operational</td>
<td></td>
</tr>
<tr>
<td>MOVE LONG</td>
<td>Length equal</td>
<td>Length low</td>
<td>Length high</td>
<td>Destructive overlap</td>
<td></td>
</tr>
<tr>
<td>MOVE LONG EXTENDED</td>
<td>Length equal</td>
<td>Length low</td>
<td>Length high</td>
<td>CPU-determined completion</td>
<td></td>
</tr>
<tr>
<td>MOVE LONG UNICODE</td>
<td>Length equal</td>
<td>Length low</td>
<td>Length high</td>
<td>CPU-determined completion</td>
<td></td>
</tr>
<tr>
<td>MOVE PAGE</td>
<td>Data moved</td>
<td>Operand 1 invalid, both valid in ES, locked, or ES error</td>
<td>Operand 2 invalid</td>
<td>--</td>
<td></td>
</tr>
</tbody>
</table>

Figure C-1 (Part 3 of 6). Summary of Condition-Code Settings
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td>MOVE STRING</td>
<td>--</td>
</tr>
<tr>
<td>MOVE TO PRIMARY</td>
<td>Length &lt;= 256</td>
</tr>
<tr>
<td>MOVE TO SECONDARY</td>
<td>Length &lt;= 256</td>
</tr>
<tr>
<td>MOVE WITH KEY</td>
<td>Length &lt;= 256</td>
</tr>
<tr>
<td>OR</td>
<td>Zero</td>
</tr>
<tr>
<td>PAGE IN</td>
<td>Page-in operation completed</td>
</tr>
<tr>
<td>PAGE OUT</td>
<td>Page-out operation completed</td>
</tr>
<tr>
<td>PERFORM LOCKED OPERATION if test bit zero</td>
<td>Equal</td>
</tr>
<tr>
<td>PERFORM LOCKED OPERATION if test bit one</td>
<td>Function code valid</td>
</tr>
<tr>
<td>RESET CHANNEL PATH</td>
<td>Function initiated</td>
</tr>
<tr>
<td>STORE CHANNEL REPORT WORD</td>
<td>CRW stored</td>
</tr>
<tr>
<td>STORE CLOCK</td>
<td>Set</td>
</tr>
<tr>
<td>STORE CLOCK EXTENDED</td>
<td>Set</td>
</tr>
<tr>
<td>STORE SUBCHANNEL</td>
<td>SCHIB stored</td>
</tr>
<tr>
<td>STORE SYSTEM INFORMATION</td>
<td>Information provided</td>
</tr>
</tbody>
</table>

*Figure C-1 (Part 4 of 6). Summary of Condition-Code Settings*
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBTRACT (gen)</td>
<td></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>Overflow</td>
</tr>
<tr>
<td>SUBTRACT (BFP)</td>
<td></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>NaN</td>
</tr>
<tr>
<td>SUBTRACT DECIMAL</td>
<td></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>Overflow</td>
</tr>
<tr>
<td>SUBTRACT HALFWORD</td>
<td></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>Not zero, no borrow</td>
</tr>
<tr>
<td>SUBTRACT LOGICAL</td>
<td></td>
<td>Not zero, borrow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBTRACT LOGICAL WITH BORROW</td>
<td></td>
<td>Zero, borrow</td>
<td>Not zero, borrow</td>
<td>Zero, no borrow</td>
<td>Not zero, no borrow</td>
</tr>
<tr>
<td>SUBTRACT NORMALIZED (HFP)</td>
<td></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>--</td>
</tr>
<tr>
<td>SUBTRACT UNNORMALIZED (HFP)</td>
<td></td>
<td>Zero</td>
<td>&lt; zero</td>
<td>&gt; zero</td>
<td>--</td>
</tr>
<tr>
<td>TEST ACCESS</td>
<td></td>
<td>ALET /zerodot</td>
<td>DU access list, no exceptions</td>
<td>PS access list, no exceptions</td>
<td>ALET 1 or exceptions</td>
</tr>
<tr>
<td>TEST ADDRESSING MODE</td>
<td></td>
<td>Twenty-four bit mode</td>
<td>Thirty-one bit mode</td>
<td>--</td>
<td>Sixty-four bit mode</td>
</tr>
<tr>
<td>TEST AND SET</td>
<td></td>
<td>Left bit zero</td>
<td>Left bit one</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TEST BLOCK</td>
<td></td>
<td>Usable</td>
<td>Not usable</td>
<td>Digit invalid</td>
<td>Sign and digit invalid</td>
</tr>
<tr>
<td>TEST DATA CLASS</td>
<td></td>
<td>Zero (no match)</td>
<td>One (match)</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TEST DECIMAL</td>
<td></td>
<td>Digits and sign valid</td>
<td>Interruption code stored</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TEST PENDING INTERRUPTION</td>
<td></td>
<td>Interruption code not stored</td>
<td>Interruption code stored</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>TEST PROTECTION</td>
<td></td>
<td>Can fetch, IFR stored;</td>
<td>Can fetch, IFR stored;</td>
<td>Cannot fetch, cannot store</td>
<td>Translation not available</td>
</tr>
<tr>
<td>TEST SUBCHANNEL</td>
<td></td>
<td>IFR stored;</td>
<td>IFR stored;</td>
<td>--</td>
<td>Not operational</td>
</tr>
<tr>
<td>TEST UNDER MASK</td>
<td></td>
<td>subchannel status-pending</td>
<td>subchannel status-pending</td>
<td>--</td>
<td>All ones</td>
</tr>
<tr>
<td>TEST UNDER MASK (HIGH/LOW)</td>
<td></td>
<td>All zeros</td>
<td>Mixed</td>
<td>Mixed, left bit one</td>
<td>All ones</td>
</tr>
<tr>
<td>TRANSLATE AND TEST</td>
<td></td>
<td>All zeros</td>
<td>Incomplete</td>
<td>Complete</td>
<td>--</td>
</tr>
</tbody>
</table>

*Figure C-1 (Part 5 of 6). Summary of Condition-Code Settings*
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSLATE AND TEST REVERSED</td>
<td>All zeros Data processed</td>
</tr>
<tr>
<td>TRANSLATE EXTENDED</td>
<td>Incomplete Op1 byte equal test byte</td>
</tr>
<tr>
<td>TRANSLATE ONE TO ONE, ONE TO TWO, TWO TO ONE, TWO</td>
<td>Character equal test character not found</td>
</tr>
<tr>
<td>UNPACK ASCII</td>
<td>Sign plus</td>
</tr>
<tr>
<td>UNPACK UNICODE</td>
<td>Sign plus</td>
</tr>
<tr>
<td>UPDATE TREE</td>
<td>Equal</td>
</tr>
<tr>
<td>ZERO AND ADD</td>
<td>Zero</td>
</tr>
<tr>
<td></td>
<td>Not equal or no comparison</td>
</tr>
<tr>
<td></td>
<td>&gt; zero</td>
</tr>
</tbody>
</table>

Explanation:

- > zero Result greater than zero.
- < zero Result less than zero.
- == 256 Equal to, or less than, 256.
- > 256 Greater than 256.
- gen General instruction.
- BFP Binary-floating-point instruction.
- High First operand high.
- HFP Hexadecimal-floating-point instruction.
- Low First operand low.
- Length Length of first operand.
- NaN Not-a-number.
- OCB Operand-control bit.

Figure C-1 (Part 6 of 6). Summary of Condition-Code Settings
## Appendix G. Table of Powers of 2

<table>
<thead>
<tr>
<th>PLUS</th>
<th>MINUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.500</td>
</tr>
<tr>
<td>2</td>
<td>0.250</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
</tr>
<tr>
<td>8</td>
<td>0.0625</td>
</tr>
<tr>
<td>16</td>
<td>0.03125</td>
</tr>
<tr>
<td>32</td>
<td>0.015625</td>
</tr>
<tr>
<td>64</td>
<td>0.0078125</td>
</tr>
<tr>
<td>128</td>
<td>0.00390625</td>
</tr>
<tr>
<td>256</td>
<td>0.001953125</td>
</tr>
<tr>
<td>512</td>
<td>0.0009765625</td>
</tr>
<tr>
<td>1,024</td>
<td>0.00048828125</td>
</tr>
<tr>
<td>2,048</td>
<td>0.000244140625</td>
</tr>
<tr>
<td>4,096</td>
<td>0.0001220703125</td>
</tr>
<tr>
<td>8,192</td>
<td>0.00006103515625</td>
</tr>
<tr>
<td>16,384</td>
<td>0.000030517578125</td>
</tr>
<tr>
<td>32,768</td>
<td>0.0000152587890625</td>
</tr>
<tr>
<td>65,536</td>
<td>0.00000762939453125</td>
</tr>
<tr>
<td>131,072</td>
<td>0.000003814697265625</td>
</tr>
<tr>
<td>262,144</td>
<td>0.0000019073486328125</td>
</tr>
<tr>
<td>524,288</td>
<td>0.00000095367431640625</td>
</tr>
<tr>
<td>1,048,576</td>
<td>0.000000476837158203125</td>
</tr>
<tr>
<td>2,097,152</td>
<td>0.0000002384185791015625</td>
</tr>
<tr>
<td>4,194,304</td>
<td>0.00000011920926855078125</td>
</tr>
<tr>
<td>8,388,608</td>
<td>0.000000059604644775390625</td>
</tr>
<tr>
<td>16,777,216</td>
<td>0.000000029802322876903125</td>
</tr>
<tr>
<td>33,554,432</td>
<td>0.000000014901161939875625</td>
</tr>
<tr>
<td>67,108,864</td>
<td>0.000000007450580594628573125</td>
</tr>
<tr>
<td>134,217,728</td>
<td>0.0000000037252902984619140625</td>
</tr>
<tr>
<td>268,435,456</td>
<td>0.00000000186266414923095703125</td>
</tr>
<tr>
<td>536,870,912</td>
<td>0.000000000888638656841843025125</td>
</tr>
<tr>
<td>1,073,741,824</td>
<td>0.000000000444747580664416875390625</td>
</tr>
<tr>
<td>2,147,483,648</td>
<td>0.0000000002223736248035678765625</td>
</tr>
<tr>
<td>4,294,967,296</td>
<td>0.000000000111368837272616029739798828125</td>
</tr>
<tr>
<td>8,589,934,592</td>
<td>0.00000000005575673047409060315625</td>
</tr>
<tr>
<td>17,179,869,184</td>
<td>0.00000000002776848623704681578125</td>
</tr>
<tr>
<td>34,359,738,368</td>
<td>0.00000000001388422107144285714292</td>
</tr>
</tbody>
</table>
Appendix H. Hexadecimal Tables

The following tables aid in converting hexadecimal values to decimal values, or the reverse.

Direct Conversion Table

This table provides direct conversion of decimal and hexadecimal numbers in these ranges:

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 to FFF</td>
<td>0000 to 4095</td>
</tr>
</tbody>
</table>

To convert numbers outside these ranges, and to convert fractions, use the hexadecimal and decimal conversion tables that follow the direct conversion table in this appendix.
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1824</td>
<td>1825</td>
<td>1826</td>
<td>1827</td>
<td>1828</td>
<td>1829</td>
<td>1830</td>
<td>1831</td>
<td>1832</td>
<td>1833</td>
<td>1834</td>
<td>1835</td>
<td>1836</td>
<td>1837</td>
<td>1838</td>
</tr>
<tr>
<td>41</td>
<td>1840</td>
<td>1841</td>
<td>1842</td>
<td>1843</td>
<td>1844</td>
<td>1845</td>
<td>1846</td>
<td>1847</td>
<td>1848</td>
<td>1849</td>
<td>1850</td>
<td>1851</td>
<td>1852</td>
<td>1853</td>
<td>1854</td>
</tr>
<tr>
<td>42</td>
<td>1856</td>
<td>1857</td>
<td>1858</td>
<td>1859</td>
<td>1860</td>
<td>1861</td>
<td>1862</td>
<td>1863</td>
<td>1864</td>
<td>1865</td>
<td>1866</td>
<td>1867</td>
<td>1868</td>
<td>1869</td>
<td>1870</td>
</tr>
<tr>
<td>43</td>
<td>1872</td>
<td>1873</td>
<td>1874</td>
<td>1875</td>
<td>1876</td>
<td>1877</td>
<td>1878</td>
<td>1879</td>
<td>1880</td>
<td>1881</td>
<td>1882</td>
<td>1883</td>
<td>1884</td>
<td>1885</td>
<td>1886</td>
</tr>
<tr>
<td>44</td>
<td>1888</td>
<td>1889</td>
<td>1890</td>
<td>1891</td>
<td>1892</td>
<td>1893</td>
<td>1894</td>
<td>1895</td>
<td>1896</td>
<td>1897</td>
<td>1898</td>
<td>1899</td>
<td>1900</td>
<td>1901</td>
<td>1902</td>
</tr>
<tr>
<td>45</td>
<td>1904</td>
<td>1905</td>
<td>1906</td>
<td>1907</td>
<td>1908</td>
<td>1909</td>
<td>1910</td>
<td>1911</td>
<td>1912</td>
<td>1913</td>
<td>1914</td>
<td>1915</td>
<td>1916</td>
<td>1917</td>
<td>1918</td>
</tr>
<tr>
<td>46</td>
<td>1920</td>
<td>1921</td>
<td>1922</td>
<td>1923</td>
<td>1924</td>
<td>1925</td>
<td>1926</td>
<td>1927</td>
<td>1928</td>
<td>1929</td>
<td>1930</td>
<td>1931</td>
<td>1932</td>
<td>1933</td>
<td>1934</td>
</tr>
<tr>
<td>53</td>
<td>2032</td>
<td>2033</td>
<td>2034</td>
<td>2035</td>
<td>2036</td>
<td>2037</td>
<td>2038</td>
<td>2039</td>
<td>2040</td>
<td>2041</td>
<td>2042</td>
<td>2043</td>
<td>2044</td>
<td>2045</td>
<td>2046</td>
</tr>
</tbody>
</table>

Appendix H. Hexadecimal Tables
<table>
<thead>
<tr>
<th>C0_</th>
<th>C1_</th>
<th>C2_</th>
<th>C3_</th>
<th>C4_</th>
<th>C5_</th>
<th>C6_</th>
<th>C7_</th>
</tr>
</thead>
<tbody>
<tr>
<td>3072</td>
<td>3073</td>
<td>3074</td>
<td>3075</td>
<td>3076</td>
<td>3077</td>
<td>3078</td>
<td>3079</td>
</tr>
<tr>
<td>C8_</td>
<td>C9_</td>
<td>CA_</td>
<td>CB_</td>
<td>CC_</td>
<td>CD_</td>
<td>CE_</td>
<td>CF_</td>
</tr>
<tr>
<td>3088</td>
<td>3089</td>
<td>3090</td>
<td>3091</td>
<td>3092</td>
<td>3093</td>
<td>3094</td>
<td>3095</td>
</tr>
<tr>
<td>C0_</td>
<td>C1_</td>
<td>C2_</td>
<td>C3_</td>
<td>C4_</td>
<td>C5_</td>
<td>C6_</td>
<td>C7_</td>
</tr>
<tr>
<td>3104</td>
<td>3105</td>
<td>3106</td>
<td>3107</td>
<td>3108</td>
<td>3109</td>
<td>3110</td>
<td>3111</td>
</tr>
<tr>
<td>D0_</td>
<td>D1_</td>
<td>D2_</td>
<td>D3_</td>
<td>D4_</td>
<td>D5_</td>
<td>D6_</td>
<td>D7_</td>
</tr>
<tr>
<td>3120</td>
<td>3121</td>
<td>3122</td>
<td>3123</td>
<td>3124</td>
<td>3125</td>
<td>3126</td>
<td>3127</td>
</tr>
<tr>
<td>D8_</td>
<td>D9_</td>
<td>DA_</td>
<td>DB_</td>
<td>DC_</td>
<td>DD_</td>
<td>DE_</td>
<td>DF_</td>
</tr>
<tr>
<td>3148</td>
<td>3149</td>
<td>3150</td>
<td>3151</td>
<td>3152</td>
<td>3153</td>
<td>3154</td>
<td>3155</td>
</tr>
<tr>
<td>E0_</td>
<td>E1_</td>
<td>E2_</td>
<td>E3_</td>
<td>E4_</td>
<td>E5_</td>
<td>E6_</td>
<td>E7_</td>
</tr>
<tr>
<td>3176</td>
<td>3177</td>
<td>3178</td>
<td>3179</td>
<td>3180</td>
<td>3181</td>
<td>3182</td>
<td>3183</td>
</tr>
<tr>
<td>F0_</td>
<td>F1_</td>
<td>F2_</td>
<td>F3_</td>
<td>F4_</td>
<td>F5_</td>
<td>F6_</td>
<td>F7_</td>
</tr>
<tr>
<td>3204</td>
<td>3205</td>
<td>3206</td>
<td>3207</td>
<td>3208</td>
<td>3209</td>
<td>3210</td>
<td>3211</td>
</tr>
<tr>
<td>G0_</td>
<td>G1_</td>
<td>G2_</td>
<td>G3_</td>
<td>G4_</td>
<td>G5_</td>
<td>G6_</td>
<td>G7_</td>
</tr>
<tr>
<td>3232</td>
<td>3233</td>
<td>3234</td>
<td>3235</td>
<td>3236</td>
<td>3237</td>
<td>3238</td>
<td>3239</td>
</tr>
<tr>
<td>H0_</td>
<td>H1_</td>
<td>H2_</td>
<td>H3_</td>
<td>H4_</td>
<td>H5_</td>
<td>H6_</td>
<td>H7_</td>
</tr>
<tr>
<td>3260</td>
<td>3261</td>
<td>3262</td>
<td>3263</td>
<td>3264</td>
<td>3265</td>
<td>3266</td>
<td>3267</td>
</tr>
</tbody>
</table>
### Conversion Table: Hexadecimal and Decimal Integers

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
</tr>
<tr>
<td>3</td>
<td>768</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
</tr>
<tr>
<td>5</td>
<td>1382</td>
</tr>
<tr>
<td>6</td>
<td>1744</td>
</tr>
<tr>
<td>7</td>
<td>2008</td>
</tr>
<tr>
<td>8</td>
<td>2272</td>
</tr>
<tr>
<td>9</td>
<td>2536</td>
</tr>
</tbody>
</table>

### Example

To convert integer numbers greater than the capacity of the table, use the techniques below:

**HEXADECIMAL TO DECIMAL**

1. Locate the column of the decimal numbers corresponding to the left-most digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the units (third from the left) position.
3. Repeat step 1 for the second position of the hexadecimal (and a remainder)
4. Add the numbers selected from the table to form the decimal number.

**DECIMAL TO HEXADECIMAL**

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
2. (b) Record the hexadecimal of the column containing the selected number.
3. (c) Subtract the selected decimal from the number to be converted.
4. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal.
5. Using the remainder from step 2, repeat all of step 1 to develop the units position of the hexadecimal.
6. Combine the terms to form the hexadecimal number.

### Powers of 16 Table

<table>
<thead>
<tr>
<th>16^n</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
</tr>
<tr>
<td>4096</td>
<td>3</td>
</tr>
<tr>
<td>65536</td>
<td>4</td>
</tr>
<tr>
<td>1048576</td>
<td>5</td>
</tr>
<tr>
<td>16777216</td>
<td>6</td>
</tr>
<tr>
<td>268435456</td>
<td>7</td>
</tr>
<tr>
<td>4294967296</td>
<td>8</td>
</tr>
<tr>
<td>68719476736</td>
<td>9</td>
</tr>
</tbody>
</table>

**Decimal Values**

Example: \(268435456 \times 10^8\) = 1000000000 = \(10^9\)

---

To convert integer numbers greater than the capacity of the table, use the techniques below:

**HEXADECIMAL TO DECIMAL**

Successive cumulative multiplication from left to right, adding units position.

Example: \(3380_{16} = 3380_{10}\)

\(16 \times 3380 = 53280 \quad D = 13\)

\(16 \times 53280 = 852480 \quad E = 30\)

\(16 \times 852480 = 13639680 \quad F = 14\)

**DECIMAL TO HEXADECIMAL**

Divide and collect the remainder in reverse order.

Example: \(3380_{10} \div 16\)

\(3380 \div 16 = 211 \quad D = 13\)

\(211 \div 16 = 13 \quad B = 10\)

\(13 \div 16 = 0 \quad A = 10\)
## Conversion Table: Hexadecimal and Decimal Fractions

### BYTE

<table>
<thead>
<tr>
<th>Bits</th>
<th>0123</th>
<th>4567</th>
<th>0123</th>
<th>4567</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>Decimal</td>
<td>Hex</td>
<td>Decimal</td>
<td>Hex</td>
</tr>
<tr>
<td>.0</td>
<td>.0000</td>
<td>.00</td>
<td>.0000</td>
<td>.0000</td>
</tr>
<tr>
<td>.1</td>
<td>.0625</td>
<td>.01</td>
<td>.0039</td>
<td>.0625</td>
</tr>
<tr>
<td>.2</td>
<td>.1250</td>
<td>.02</td>
<td>.0078</td>
<td>.1250</td>
</tr>
<tr>
<td>.3</td>
<td>.1875</td>
<td>.03</td>
<td>.0117</td>
<td>1875</td>
</tr>
<tr>
<td>.4</td>
<td>.2500</td>
<td>.04</td>
<td>.0156</td>
<td>2500</td>
</tr>
<tr>
<td>.5</td>
<td>.3125</td>
<td>.05</td>
<td>.0195</td>
<td>3125</td>
</tr>
<tr>
<td>.6</td>
<td>.3750</td>
<td>.06</td>
<td>.0234</td>
<td>3750</td>
</tr>
<tr>
<td>.7</td>
<td>.4375</td>
<td>.07</td>
<td>.0273</td>
<td>4375</td>
</tr>
<tr>
<td>.8</td>
<td>.5000</td>
<td>.08</td>
<td>.0312</td>
<td>5000</td>
</tr>
<tr>
<td>.9</td>
<td>.5625</td>
<td>.09</td>
<td>.0351</td>
<td>5625</td>
</tr>
<tr>
<td>.A</td>
<td>.6250</td>
<td>.0A</td>
<td>.0390</td>
<td>6250</td>
</tr>
<tr>
<td>.B</td>
<td>.6875</td>
<td>.0B</td>
<td>.0429</td>
<td>6875</td>
</tr>
<tr>
<td>.C</td>
<td>.7500</td>
<td>.0C</td>
<td>.0468</td>
<td>7500</td>
</tr>
<tr>
<td>.D</td>
<td>.8125</td>
<td>.0D</td>
<td>.0507</td>
<td>8125</td>
</tr>
<tr>
<td>.E</td>
<td>.8750</td>
<td>.0E</td>
<td>.0546</td>
<td>8750</td>
</tr>
<tr>
<td>.F</td>
<td>.9375</td>
<td>.0F</td>
<td>.0585</td>
<td>9375</td>
</tr>
</tbody>
</table>

### TO CONVET .ABC HEXADECIMAL TO DECIMAL

Find .A in position 1 .6250

Find .B in position 2 .0429 6825

Find .CC in position 3 .0029 2968 7500

ABC is equal to .6708 9843 7500

### TO CONVET .13 DECIMAL TO HEXADECIMAL

1. Find .1250 next lowest to 1300 subtract - .1250 = .2 Hex
2. Find .0039 0625 next lowest to .0050 0000 subtract - .0039 0625 = .01
3. Find .0097 7656 2500 .0010 9375 0000 subtract - .0009 7656 2500 = .004
4. Find .0001 0681 1523 4375 .0001 1718 7500 0000 subtract - .0001 0681 1523 4375 = .0007 .0000 1037 5976 5625 = .2147 Hex

5. .13 Decimal is approximately equal to .2147 Hex

To convert fractions beyond the capacity of the table, use techniques below:

### HEXADECIMAL TO FRACTION DECIMAL

Convert the hexadecimal fraction to its decimal equivalent using the same technique as for integer numbers. Divide the results by 16^n (n is the number of fraction positions).

Example: .8A7 = .540771e

8A7e = 2215e = .540771e

16^3 = 4096 4096 2215.000000

### DECIMAL FRACTION TO HEXADECIMAL

Collect integer parts of product in the order of calculation.

Example: .5408e = .8A7e

8 A

.6528

10 .4448

7 .1168

Appendix H. Hexadecimal Tables H-7
### Hexadecimal Addition and Subtraction Table

Example: $6 + 2 = 8$, $8 - 2 = 6$, and $8 - 6 = 2$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>1A</td>
</tr>
<tr>
<td>C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>1A</td>
<td>1B</td>
</tr>
<tr>
<td>D</td>
<td>0E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>1A</td>
<td>1B</td>
<td>1C</td>
</tr>
<tr>
<td>E</td>
<td>0F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>1A</td>
<td>1B</td>
<td>1C</td>
<td>1D</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>1A</td>
<td>1B</td>
<td>1C</td>
<td>1D</td>
<td>1E</td>
</tr>
</tbody>
</table>

### Hexadecimal Multiplication Table

Example: $2 \times 4 = 08$, $F \times 2 = 1E$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0A</td>
<td>0B</td>
<td>0C</td>
<td>0D</td>
<td>0E</td>
<td>0F</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>04</td>
<td>06</td>
<td>08</td>
<td>0A</td>
<td>0C</td>
<td>0E</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>1A</td>
<td>1C</td>
<td>1E</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>06</td>
<td>09</td>
<td>0C</td>
<td>0F</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>1B</td>
<td>1E</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>2A</td>
<td>2D</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>08</td>
<td>0C</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>1C</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>2C</td>
<td>30</td>
<td>34</td>
<td>38</td>
<td>3C</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>0A</td>
<td>0F</td>
<td>14</td>
<td>19</td>
<td>1E</td>
<td>23</td>
<td>28</td>
<td>2D</td>
<td>32</td>
<td>37</td>
<td>41</td>
<td>46</td>
<td>4B</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>0C</td>
<td>12</td>
<td>18</td>
<td>1E</td>
<td>24</td>
<td>2A</td>
<td>30</td>
<td>36</td>
<td>3C</td>
<td>42</td>
<td>48</td>
<td>4E</td>
<td>54</td>
<td>5A</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>0E</td>
<td>15</td>
<td>1C</td>
<td>23</td>
<td>2A</td>
<td>31</td>
<td>38</td>
<td>3F</td>
<td>46</td>
<td>4D</td>
<td>54</td>
<td>5B</td>
<td>62</td>
<td>69</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>10</td>
<td>18</td>
<td>20</td>
<td>28</td>
<td>30</td>
<td>38</td>
<td>40</td>
<td>48</td>
<td>50</td>
<td>58</td>
<td>60</td>
<td>68</td>
<td>70</td>
<td>78</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>12</td>
<td>1B</td>
<td>24</td>
<td>2D</td>
<td>36</td>
<td>3F</td>
<td>48</td>
<td>51</td>
<td>5A</td>
<td>63</td>
<td>6C</td>
<td>75</td>
<td>7E</td>
<td>87</td>
</tr>
<tr>
<td>A</td>
<td>0A</td>
<td>14</td>
<td>1E</td>
<td>28</td>
<td>32</td>
<td>3C</td>
<td>46</td>
<td>50</td>
<td>5A</td>
<td>64</td>
<td>6E</td>
<td>78</td>
<td>82</td>
<td>8C</td>
<td>96</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>16</td>
<td>21</td>
<td>2C</td>
<td>37</td>
<td>42</td>
<td>4D</td>
<td>58</td>
<td>63</td>
<td>6E</td>
<td>79</td>
<td>84</td>
<td>8F</td>
<td>9A</td>
<td>A5</td>
</tr>
<tr>
<td>C</td>
<td>0C</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>3C</td>
<td>48</td>
<td>54</td>
<td>60</td>
<td>6C</td>
<td>78</td>
<td>84</td>
<td>90</td>
<td>9C</td>
<td>A8</td>
<td>B4</td>
</tr>
<tr>
<td>D</td>
<td>0D</td>
<td>1A</td>
<td>27</td>
<td>34</td>
<td>41</td>
<td>4E</td>
<td>58</td>
<td>68</td>
<td>75</td>
<td>82</td>
<td>8F</td>
<td>9C</td>
<td>A9</td>
<td>B6</td>
<td>C3</td>
</tr>
<tr>
<td>E</td>
<td>0E</td>
<td>1C</td>
<td>2A</td>
<td>38</td>
<td>46</td>
<td>54</td>
<td>62</td>
<td>70</td>
<td>7E</td>
<td>8C</td>
<td>9A</td>
<td>A8</td>
<td>B6</td>
<td>C4</td>
<td>D2</td>
</tr>
<tr>
<td>F</td>
<td>0F</td>
<td>1E</td>
<td>2D</td>
<td>3C</td>
<td>48</td>
<td>5A</td>
<td>69</td>
<td>78</td>
<td>87</td>
<td>96</td>
<td>A5</td>
<td>B4</td>
<td>C3</td>
<td>D2</td>
<td>E1</td>
</tr>
</tbody>
</table>
Appendix I. EBCDIC and Other Codes

The following table shows the Extended Binary-Coded-Decimal Interchange Code (EBCDIC) and other codes. Details may be found in the notes on page I-4.

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>EBCDIC</th>
<th>AS-ISO (1)</th>
<th>CII -6 IBM-PC</th>
<th>BookMaster Symbol Names(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>NUL</td>
<td>NUL NUL NUL</td>
<td>FACE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>SOH</td>
<td>SOH SOH SOH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>02</td>
<td>STX</td>
<td>STX STX STX</td>
<td>FACE</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>03</td>
<td>ETX</td>
<td>ETX ETX ETX</td>
<td>HEART</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>04</td>
<td>SEL</td>
<td>EOT EOT EOT</td>
<td>DIAMOND</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>05</td>
<td>HT</td>
<td>ENQ ENQ ENQ</td>
<td>CLUB</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>06</td>
<td>RNL</td>
<td>ACK ACK ACK</td>
<td>SPADE</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>07</td>
<td>DEL</td>
<td>BEL BEL BEL</td>
<td>bullet</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>08</td>
<td>GE</td>
<td>BS BS BS</td>
<td>revbulate</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>09</td>
<td>SP</td>
<td>HT HT HT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>RPT</td>
<td>LF LF LF</td>
<td>recuir</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>VT</td>
<td>VT VT VT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>FF</td>
<td>FF FF FF FF</td>
<td>female</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>CR</td>
<td>CR CR CR</td>
<td>note18</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>SO</td>
<td>SO SO SO SO</td>
<td>note16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>SI</td>
<td>SI SI SI</td>
<td>sun</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>DLE</td>
<td>DLE DLE DLE</td>
<td>rahread</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>DC1</td>
<td>DC1 DC1 DC1</td>
<td>lahead</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>DC2</td>
<td>DC2 DC2 DC2</td>
<td>udarrow</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>DC3</td>
<td>DC3 DC3 DC3</td>
<td>dbxclam</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>RES/ENP</td>
<td>DC4 DC4 DC4</td>
<td>par</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>NL</td>
<td>NAK NAK NAK</td>
<td>section</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>BS</td>
<td>SYN SYN SYN</td>
<td>overline</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>POC</td>
<td>ETB ETB ETB</td>
<td>udarrowwus</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>CAN</td>
<td>CAN CAN CAN</td>
<td>narrow</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>EM</td>
<td>EM EM EM</td>
<td>darrow</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>UBS</td>
<td>SUB SUB IFS</td>
<td>narrow</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>CU1</td>
<td>ESC ESC ESC</td>
<td>narrow</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>IFS</td>
<td>FS IFS DEL</td>
<td>inotusd</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>IGS</td>
<td>GS IGS IGS</td>
<td>lrarrow</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>IRS</td>
<td>RS IRS RS RS</td>
<td>urlhead</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>ITB/IUS</td>
<td>US IUS US US</td>
<td>dahhead</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control-Character Representations
ACK Acknowledge
BEL Bell
BS Backspace
BYP Bypass
CAN Cancel
CR Carriage Return
CSP Control Sequence Prefix
CU1 Customer Use 1
CU3 Customer Use 3
DC1 Device Control 1
DC2 Device Control 2
DC3 Device Control 3
DC4 Device Control 4
DEL Delete
DLE Data Link Escape
DS Digit Select
EM End of Medium
ENQ Enable Presentation
EO End of Transmission
ESC Escape
ETB End of Transmission Block
ETX End of Text
FF Form Feed
FS Field Separator
GE Graphic Escape
HT Horizontal Tab
IFS Interchange File Separator
IGS Interchange Group Separator
INP Inhibit Presentation
IR Indext Return
IRS Interchange Record Separator
IT Indent Tab
ITB Intermediate Transmission
IUS International Unit Separator
LE Line Feed
MFA Modify Field Attribute
MM Modify Mode
NL New Line
NUL Null
OC Open Close
POC Program-Operator
PP Presentation Position
RES Restore
RFF Required Form Feed
RNL Required New Line
RPT Repeat
RT Return
SA Set Attribute
SB Subscript
SEL Select
SIGN Significant
SBS Superscript
SP Sub
STX Start of Text
SYN Synchronous Idle
TAB Tabulator
TRN Transparent
UES Unit Backspace
VT Vertical Tab
WUS Word Underscore

Formatting-Character Representations
NSP Numeric Space
SP Space
RSP Required Space
SHY Syllable Hyphen

© Copyright IBM Corp. 1990-2004 I-1
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Symbol</th>
<th>EBCDIC(4)</th>
<th>AS- ISO IBM</th>
<th>BookMaster Symbol Names(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>040</td>
<td>SP SP SP SP SP</td>
<td>@ @ @</td>
<td>antsigin</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>041</td>
<td>RSP RSP RSP RSP RSP</td>
<td>A A A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>042</td>
<td>a a a B B B</td>
<td>ac</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>043</td>
<td>a a a C C C</td>
<td>aee</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 68  | 044 | a a a D D D | ag | |
| 69  | 045 | a a a E E E | aa | |
| 70  | 046 | a a a F F F | at | |
| 71  | 047 | a a a G G G | ao | |

| 72  | 048 | c c c H H H | cc | |
| 73  | 049 | n n n I I I | nt | |
| 74  | 050 | ¥ ¥ ¥ | yen | |
| 75  | 051 | $ $ $ | dollar | |

| 76  | 06C | ë ë ë | S S S | ee |
| 77  | 06D | é é é | Q Q Q | ea |

| 81  | 07A | Â Â Â | £ pound sterling | pound sterling |
| 82  | 07B | Æ Æ Æ | € euro | euro |
| 83  | 07C | Ñ Ñ Ñ | ¥ yen | yen |
| 84  | 07D | ç ç ç | cent, lbrk | cent, lbrk |
| 85  | 07E | à à à | grave | grave |
| 86  | 07F | â â â | tilde | tilde |

| 88  | 08C | Ñ Ñ Ñ | iacute | iacute |
| 89  | 08D | ç ç ç | ccedilla | ccedilla |
| 90  | 08E | Ë Ë Ë | Egrave | Egrave |
| 91  | 08F | Ê Ê Ê | Ecircumflex | Ecircumflex |

| 92  | 09C | ✠ ✠ ✠ | asterisk, blaslash | asterisk, blaslash |
| 93  | 09D | ✠ ✠ ✠ | rpar, rbrk | rpar, rbrk |
| 94  | 09E | ✠ ✠ ✠ | semi, hat | semi, hat |
| 95  | 09F | ✠ ✠ ✠ | inot, hat, us | inot, hat, us |

---

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Symbol</th>
<th>EBCDIC(4)</th>
<th>AS- ISO IBM</th>
<th>BookMaster Symbol Names(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>140</td>
<td>- - - - -</td>
<td>- - - - -</td>
<td>hyphen or minus, grave</td>
<td>hyphen or minus, grave</td>
</tr>
<tr>
<td>97</td>
<td>141</td>
<td>/ / / / /</td>
<td>a a a</td>
<td>divslash or slash</td>
<td>divslash or slash</td>
</tr>
<tr>
<td>98</td>
<td>142</td>
<td>A A A b b b</td>
<td>b b b</td>
<td>Ac</td>
<td>Ac</td>
</tr>
<tr>
<td>99</td>
<td>143</td>
<td>A A A c c c</td>
<td>c c c</td>
<td>Ae</td>
<td>Ae</td>
</tr>
</tbody>
</table>

| 100 | 150 | A A A d d d | d d d | Ag | Ag |
| 101 | 151 | A A A e e e | e e e | Aa | Aa |
| 102 | 152 | A A A f f f | f f f | At | At |
| 103 | 153 | A A A g g g | g g g | Ao | Ao |

| 104 | 154 | c c c h h h | Cc | Cc |
| 105 | 155 | N N N i i i | Nt | Nt |
| 106 | 156 | i i i i j j j | lbrk | lbrk |
| 107 | 157 | . . . . | comma | comma |

---

**BookMaster Symbols for Character Set 0697 (See Note (4))**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Symbol</th>
<th>Description</th>
<th>Symbol</th>
<th>Description</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>at sign</td>
<td>#</td>
<td>number sign</td>
<td>%</td>
<td>percent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>dollar</td>
<td>&amp;</td>
<td>amp</td>
<td>/</td>
<td>divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>lpar</td>
<td>)</td>
<td>rpar</td>
<td>[</td>
<td>lbrk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>lbrc</td>
<td>}</td>
<td>rbrc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. EBCDIC(4) represents the binary-coded decimal inter-character code used in IBM mainframe systems.
2. AS-ISO IBM symbol names are the standard identifiers for symbols used in IBM's international standards.
3. BookMaster symbol names are used for compatibility with IBM's BookMaster software.
4. The Note (4) refers to specific conditions or exceptions that apply to the symbols listed.
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>EBCDIC(4) BIC 94C 037 500 1047</th>
<th>ISO IBM-PC -B 437 850</th>
<th>BookMaster Symbol Names(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>80</td>
<td>☛ ☛ ☛</td>
<td>Cs, Cs</td>
<td>ACP</td>
</tr>
<tr>
<td>129</td>
<td>81</td>
<td>a a a a a a a a</td>
<td>ü ü</td>
<td>ACP</td>
</tr>
<tr>
<td>130</td>
<td>82</td>
<td>b b b b b b b b</td>
<td>BPH é é</td>
<td>ACP</td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td>c c c c c c c c</td>
<td>NBH å å</td>
<td>ACP</td>
</tr>
<tr>
<td>132</td>
<td>84</td>
<td>d d d d d d d d</td>
<td>IND å å</td>
<td>ACP</td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td>e e e e e e e e</td>
<td>NEL å å</td>
<td>ACP</td>
</tr>
<tr>
<td>134</td>
<td>86</td>
<td>f f f f f f f f</td>
<td>SSA å å</td>
<td>ACP</td>
</tr>
<tr>
<td>135</td>
<td>87</td>
<td>g g g g g g g g</td>
<td>ESA ç ç</td>
<td>ACP</td>
</tr>
<tr>
<td>136</td>
<td>88</td>
<td>h h h h h h h h</td>
<td>HTS å å</td>
<td>ACP</td>
</tr>
<tr>
<td>137</td>
<td>89</td>
<td>i i i i i i i i</td>
<td>HTJ å å</td>
<td>ACP</td>
</tr>
<tr>
<td>139</td>
<td>8B</td>
<td>l l l l l l l l</td>
<td>STS å å</td>
<td>ACP</td>
</tr>
<tr>
<td>140</td>
<td>8C</td>
<td>» » » » » » » »</td>
<td>SM5 ® ®</td>
<td>ACP</td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td>¶ ¶ ¶ ¶ ¶ ¶ ¶ ¶</td>
<td>par, bx2/zerodot/zerodot12, Ac</td>
<td>ACP</td>
</tr>
<tr>
<td>142</td>
<td>8E</td>
<td>§ § § § § § § §</td>
<td>section, mu(6), bx2/zerodot/zerodot21, Ac</td>
<td>ACP</td>
</tr>
<tr>
<td>143</td>
<td>8F</td>
<td>¶ ¶ ¶ ¶ ¶ ¶ ¶ ¶</td>
<td>PLD å å</td>
<td>ACP</td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td>° ° ° ° ° ° ° °</td>
<td>DCS É É degree, Ea</td>
<td>ACP</td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td>j j j j j j j j</td>
<td>PUI æ æ</td>
<td>ACP</td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td>k k k k k k k k</td>
<td>POU Æ Æ</td>
<td>ACP</td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td>l l l l l l l l</td>
<td>STS Æ Æ</td>
<td>ACP</td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td>m m m m m m m m</td>
<td>CCH ö ö</td>
<td>ACP</td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td>n n n n n n n n</td>
<td>MW ö ö</td>
<td>ACP</td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td>o o o o o o o o</td>
<td>SPA ø ø</td>
<td>ACP</td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td>p p p p p p p p</td>
<td>EPA ø ø</td>
<td>ACP</td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td>q q q q q q q q</td>
<td>SOS ö ö</td>
<td>ACP</td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td>r r r r r r r r</td>
<td>Oe</td>
<td>ACP</td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td>e e e e e e e e</td>
<td>SCI Û Û</td>
<td>ACP</td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td>æ æ æ æ æ æ æ æ</td>
<td>aus, cent, os</td>
<td>ACP</td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td>æ æ æ æ</td>
<td>ST £ £</td>
<td>ACP</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>æ æ æ æ</td>
<td>OSC ¥ Ø</td>
<td>ACP</td>
</tr>
<tr>
<td>158</td>
<td>9E</td>
<td>æ æ æ æ</td>
<td>PM H X</td>
<td>ACP</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>æ æ æ æ</td>
<td>ACP f f</td>
<td>currency, fnof(5)</td>
</tr>
</tbody>
</table>

### Additional ISO-8 Control-Character Representations

<table>
<thead>
<tr>
<th>APC Application Program</th>
<th>HTS Character Tabulation Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPH Break Permitted Here</td>
<td>IFS Information Separator Four</td>
</tr>
<tr>
<td>CCH Cancel Character</td>
<td>IND Index</td>
</tr>
<tr>
<td>CSI Control Sequence Introducer</td>
<td>IRS Information Separator Two</td>
</tr>
<tr>
<td>DCS Device Control String</td>
<td>MW Message Waiting</td>
</tr>
<tr>
<td>EPA End of Guarded Area</td>
<td>NBH No Break Here</td>
</tr>
<tr>
<td>ESA End of Selected Area</td>
<td>NEL Next Line</td>
</tr>
<tr>
<td>HTJ Character Tabulation with Justification</td>
<td>OSC Operating System Command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>EBCDIC(4) BIC 94C 037 500 1047</th>
<th>ISO IBM-PC -B 437 850</th>
<th>BookMaster Symbol Names(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>A0</td>
<td>µ µ µ</td>
<td>RSP å å</td>
<td>ACP</td>
</tr>
<tr>
<td>161</td>
<td>A1</td>
<td>µ µ µ</td>
<td>i i i</td>
<td>ACP</td>
</tr>
<tr>
<td>162</td>
<td>A2</td>
<td>ss ss ss ss</td>
<td>Æ Æ</td>
<td>cent, ao</td>
</tr>
<tr>
<td>163</td>
<td>A3</td>
<td>t t t t t t</td>
<td>É É É</td>
<td>Lsterling, ua</td>
</tr>
<tr>
<td>164</td>
<td>A4</td>
<td>û û û û</td>
<td>currency, nt</td>
<td>ACP</td>
</tr>
<tr>
<td>165</td>
<td>A5</td>
<td>ï ï ï ï</td>
<td>yen, Nt</td>
<td>ACP</td>
</tr>
<tr>
<td>166</td>
<td>A6</td>
<td>ÿ ÿ ÿ ÿ</td>
<td>splitvbar, aus</td>
<td>ACP</td>
</tr>
<tr>
<td>167</td>
<td>A7</td>
<td>ÿ ÿ ÿ ÿ</td>
<td>section, aus</td>
<td>ACP</td>
</tr>
<tr>
<td>168</td>
<td>A8</td>
<td>û û û û</td>
<td>umlaut, invq</td>
<td>ACP</td>
</tr>
<tr>
<td>169</td>
<td>A9</td>
<td>ã ã ã ã</td>
<td>copyr, lnnotrev, regtm</td>
<td>ACP</td>
</tr>
<tr>
<td>170</td>
<td>AA</td>
<td>ï ï ï ï</td>
<td>inv, aus, lnnot</td>
<td>ACP</td>
</tr>
<tr>
<td>171</td>
<td>AB</td>
<td>ð ð ð ð</td>
<td>invq, odqf, frac12</td>
<td>ACP</td>
</tr>
<tr>
<td>172</td>
<td>AC</td>
<td>ð ð ð ð</td>
<td>Dstroke or Eth, lnnot, frac14</td>
<td>ACP</td>
</tr>
<tr>
<td>173</td>
<td>AD</td>
<td>ð ð ð ð</td>
<td>ya, lbrk, inv, lnnot</td>
<td>ACP</td>
</tr>
<tr>
<td>174</td>
<td>AE</td>
<td>ð ð ð ð</td>
<td>thor, regtm, odqf</td>
<td>ACP</td>
</tr>
<tr>
<td>175</td>
<td>AF</td>
<td>ð ð ð ð</td>
<td>regtm, overline, cdqf</td>
<td>ACP</td>
</tr>
<tr>
<td>176</td>
<td>A0</td>
<td>ð ð ð ð</td>
<td>hat, cent, lnnot, degree, box14</td>
<td>ACP</td>
</tr>
<tr>
<td>177</td>
<td>B1</td>
<td>ð ð ð ð</td>
<td>Lsterling, pm, box12</td>
<td>ACP</td>
</tr>
<tr>
<td>178</td>
<td>B2</td>
<td>ð ð ð ð</td>
<td>yen, sup2, box34</td>
<td>ACP</td>
</tr>
<tr>
<td>179</td>
<td>B3</td>
<td>ð ð ð ð</td>
<td>smultdot, sup3, boxv</td>
<td>ACP</td>
</tr>
<tr>
<td>180</td>
<td>B4</td>
<td>ð ð ð ð</td>
<td>copyr, acute, bxrj</td>
<td>ACP</td>
</tr>
<tr>
<td>181</td>
<td>B5</td>
<td>ð ð ð ð</td>
<td>section, mu(6), bx1012, Aa</td>
<td>ACP</td>
</tr>
<tr>
<td>182</td>
<td>B6</td>
<td>ð ð ð ð</td>
<td>par, bx2021, Ac</td>
<td>ACP</td>
</tr>
<tr>
<td>183</td>
<td>B7</td>
<td>ð ð ð ð</td>
<td>frac14, smultdot, bx0021, Ag</td>
<td>ACP</td>
</tr>
<tr>
<td>184</td>
<td>B8</td>
<td>ð ð ð ð</td>
<td>frac12, cedilla, bx0012, copyr</td>
<td>ACP</td>
</tr>
<tr>
<td>185</td>
<td>B9</td>
<td>ð ð ð ð</td>
<td>frac34, sup1, bx2022</td>
<td>ACP</td>
</tr>
<tr>
<td>186</td>
<td>BA</td>
<td>ð ð ð ð</td>
<td>lbrk, lnnot, Ye, ous, bx2020</td>
<td>ACP</td>
</tr>
<tr>
<td>187</td>
<td>BB</td>
<td>ð ð ð ð</td>
<td>rbrk, vbar, umlaut, cdqf, bx0022</td>
<td>ACP</td>
</tr>
<tr>
<td>188</td>
<td>BC</td>
<td>ð ð ð ð</td>
<td>overline, frac14, bx2002</td>
<td>ACP</td>
</tr>
<tr>
<td>189</td>
<td>BD</td>
<td>ð ð ð ð</td>
<td>umlaut, rbrk, frac12, bx2001, cent</td>
<td>ACP</td>
</tr>
<tr>
<td>190</td>
<td>BE</td>
<td>ð ð ð ð</td>
<td>acute, frac34, bx1002, yen</td>
<td>ACP</td>
</tr>
<tr>
<td>191</td>
<td>BF</td>
<td>ð ð ð ð</td>
<td>mult, invq, bxur</td>
<td>ACP</td>
</tr>
</tbody>
</table>

Appendix I. EBCDIC and Other Codes
### EBCDIC(4) and BookMaster

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>EBCDIC(4)</th>
<th>ISO IBM-PC</th>
<th>BookMaster</th>
<th>Symbol Names(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>192</td>
<td>C0</td>
<td>1bc, Ag, bx11</td>
<td>Aa, bxbj</td>
<td>Ibrc, Ag, bx11</td>
<td></td>
</tr>
<tr>
<td>193</td>
<td>C1</td>
<td>A A A A A A</td>
<td>Aa, bxjA</td>
<td>Ac, bxjtj</td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>C2</td>
<td>B B B B B B</td>
<td>Ac, bxjtj</td>
<td>Ac, bxjtj</td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>C3</td>
<td>C C C C C C</td>
<td>At, bxj</td>
<td>At, bxj</td>
<td></td>
</tr>
<tr>
<td>196</td>
<td>C4</td>
<td>D D D D D D</td>
<td>Aa, bxhA</td>
<td>Ae, bxhA</td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>C5</td>
<td>E E E E E E</td>
<td>Ao, bxjE</td>
<td>Ao, bxjE</td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>C6</td>
<td>F F F F F F</td>
<td>AEIg, bx1210,</td>
<td>Ac, bx1210, At</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>C7</td>
<td>G G G G G G</td>
<td>At, bx1210, At</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>CB</td>
<td>H H H H H H</td>
<td>Eg, bx2200</td>
<td>Eg, bx2200</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>CA</td>
<td>I I I I I I</td>
<td>Ea, bx2200</td>
<td>Ea, bx2200</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>CB</td>
<td>SHY SHY SHY SHY SHY</td>
<td>Ec, bx2202</td>
<td>Ec, bx2202</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>CB</td>
<td>SHY SHY SHY SHY SHY</td>
<td>oc, Ec, bx1202</td>
<td>oc, Ec, bx1202</td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>CC</td>
<td>õ õ õ I = = I = =</td>
<td>oe, Ig, bx2220</td>
<td>oe, Ig, bx2220</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>CD</td>
<td>õ õ õ I = = I = =</td>
<td>og, Ig, bx2220</td>
<td>og, Ig, bx2220</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>CE</td>
<td>õ õ õ I = = I = =</td>
<td>oa, Ic, bx2222</td>
<td>oa, Ic, bx2222</td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>CF</td>
<td>õ õ õ I = = I = =</td>
<td>ot, ie, bx1202, currency</td>
<td>ot, ie, bx1202, currency</td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>D0</td>
<td>} } } } } } D D D</td>
<td>rbrc, Ostroke or Eth, bx2101, Eth, bx2012, Ostroke or Eth, bx2012, Eth</td>
<td>rbrc, Ostroke or Eth, bx2101, Eth, bx2012, Ostroke or Eth, bx2012, Eth</td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>D1</td>
<td>J J J J J J</td>
<td>Nt, bx2012, Eth or Ostroke, bx2101, Eth</td>
<td>Nt, bx2012, Eth or Ostroke, bx2101, Eth</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>D2</td>
<td>K K K K K K</td>
<td>Og, bx1210, Ec</td>
<td>Og, bx1210, Ec</td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>D3</td>
<td>L L L L L L</td>
<td>Oa, bx2100, Ec</td>
<td>Oa, bx2100, Ec</td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>D4</td>
<td>M M M M M M</td>
<td>Oc, bx1200, Eg</td>
<td>Oc, bx1200, Eg</td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>D5</td>
<td>N N N N N N</td>
<td>Ot, bx2010, Identless</td>
<td>Ot, bx2010, Identless</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>D6</td>
<td>O O O O O O</td>
<td>Oe, bx1202, Ia</td>
<td>Oe, bx1202, Ia</td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>D7</td>
<td>P P P P P P</td>
<td>uc, Uc, bx1201, Ic</td>
<td>uc, Uc, bx1201, Ic</td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>DB</td>
<td>Q Q Q Q Q Q</td>
<td>Os, bx1212, Ie</td>
<td>Os, bx1212, Ie</td>
<td></td>
</tr>
<tr>
<td>217</td>
<td>D9</td>
<td>R R R R R R</td>
<td>Ug, bxlr</td>
<td>Ug, bxlr</td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>DA</td>
<td>i i i i i i</td>
<td>sup1, Ia, bxul</td>
<td>sup1, Ia, bxul</td>
<td></td>
</tr>
<tr>
<td>219</td>
<td>DB</td>
<td>ã ã ã ã ã ã</td>
<td>uc, Uc, BOX</td>
<td>uc, Uc, BOX</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>DC</td>
<td>³ ³ ³ ³ ³ ³</td>
<td>ue, Ue, BOXBOT</td>
<td>ue, Ue, BOXBOT</td>
<td></td>
</tr>
<tr>
<td>221</td>
<td>DO</td>
<td>³ ³ ³ ³ ³ ³</td>
<td>ug, Ya, BOXLEFT, splittvar</td>
<td>ug, Ya, BOXLEFT, splittvar</td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>DE</td>
<td>³ ³ ³ ³ ³ ³</td>
<td>ua, thorn, BOXRIGHT, ñ</td>
<td>ua, thorn, BOXRIGHT, ñ</td>
<td></td>
</tr>
<tr>
<td>223</td>
<td>DF</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ye, ye, BOXTOP</td>
<td>ye, ye, BOXTOP</td>
<td></td>
</tr>
</tbody>
</table>

### ISO IBM-PC

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>EBCDIC(4)</th>
<th>ISO IBM-PC</th>
<th>BookMaster</th>
<th>Symbol Names(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>224</td>
<td>E0</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>E1</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
<tr>
<td>226</td>
<td>E2</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
<tr>
<td>227</td>
<td>E3</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
<tr>
<td>228</td>
<td>E4</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
<tr>
<td>229</td>
<td>E5</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>E6</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
<tr>
<td>231</td>
<td>E7</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td>ñ ñ ñ ñ ñ ñ</td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

1. The ASCII controls and graphics are from ANSI X3.4. The ISO-8 controls are from ISO 6429, and the graphics are from ISO 8859-1. The ISO-8 graphics are code page 00819, named ISO/ANSI Multilingual. IBM-PC controls and graphics are shown. The graphics are common to code page 00437, named Personal Computer, and code page 00850, named Personal Computer - Multilingual Page. Code pages 00437 and 00850 are shown separately beginning at X′80′, after which they diverge in content.
2. The symbol names shown are to be preceded by an ampersand (&) and followed by a period (.) to form a symbol. Source: IBM BookMaster User's Guide Release 4.0, SC34-5009.
3. ASCII, ISO-8, and IBM-PC X′27′ and EBCDIC X′7D′ are an apostrophe having the appearance of a straight single quote. The BookMaster "apos" produces a character having the appearance of an accent acute.
4. Five columns of EBCDIC graphics are shown. The first is the 81-character character set 0640, called the syntactic character set, that is mapped the same on all EBCDIC code pages. The second is the standard IBM 94-character character set mapped on code page 00037. The third is code page 00037, named USA/Canada - CECP (Country Extended Code Page). The fourth is code page 00500, named International #5. The fifth is code page 01047, named Latin 1/Open Systems. Code pages 00037, 00500, 01047, and 00819 (ISO-8) all map the 189-character character set 0697. Source: National Language Support Reference Manual Volume 2, SE09-8002.
5. ñ, –, and θ are of nonstandard width.
6. EBCDIC X′AO′ and ISO-8 X′BS′ are micro but resemble mu. The BookMaster "usec" produces a character of nonstandard width.

### Symbol Names:

The symbol names shown are to be preceded by an ampersand (&) and followed by a period (.) to form a symbol.
Index

Numerics
2K-IDA control  15-25
370-XA architecture  1-14

A
A (ADD) binary instruction  7-18
absolute address  3-3
absolute storage  3-3
access-control bits in storage key  3-8
access exceptions  6-36, 6-43
   priority of  6-43
   recognition of  6-36
access key  3-9
   for channel-program execution  3-9, 15-22
   for channel-subsystem monitoring  3-9
   for CPU  3-9
access list  5-54
   See also access-list entry
access-register mode  3-35
access-register translation (ART)  5-51
   as part of LOAD REAL ADDRESS, STORE REAL
   ADDRESS, TEST ACCESS, and TEST PRO-
   TECTION  5-57
   introduction to  5-45
   lookaside buffer
   See ALB
   sequence of table fetches  5-94
access-register-translation (ART) tables  5-53
access registers  2-4
   designation of  5-44
   functions of  5-44
   instructions for use of  5-51
   save areas for  3-63
   validity bit for  11-22
access to storage  5-88
   See also reference
active
   device  16-15
   subchannel  16-15
active allegiance  15-12
active communication  15-12
activity-control field (SCSW)  16-13
   following TEST SUBCHANNEL  14-20
AD (ADD NORMALIZED) HFP instruction  18-8
   example A-39
adaptive dictionary  7-80
ADB (ADD) BFP instruction  19-18
ADBR (ADD) BFP instruction  19-18
ADD BFP instructions  19-18
ADD binary instructions  7-18
ADD DECIMAL instruction  8-5
   example A-34
ADD HALFWORD IMMEDIATE instruction  7-18
ADD HALFWORD instruction
   example A-7
ADD HALFWORD instructions  7-18
ADD LOGICAL instructions  7-19
ADD LOGICAL WITH CARRY instructions  7-20
ADD NORMALIZED HFP instructions  18-8
   example A-39
ADD UNNORMALIZED HFP instructions  18-10
   example A-39
additional floating-point (AFP) registers  9-2
address  3-2
   24-bit, 31-bit, and 64-bit  3-5
   backward stack-entry  5-79
   base
   See base address
branch
   See branch address
channel-program
   See channel-program address
comparison  12-1
   controls for  12-1
   effect on CPU state  4-2

© Copyright IBM Corp. 1990-2004
address (continued)

CPU
  See CPU address
data (I/O)
  See data address
effective
  See effective address
failing-storage
  See failing-storage address
format 3-2
forward-section-header 5-80
generation 5-8
  for storage addressing 3-6
I/O 13-5
instruction
  See instruction address
invalid 6-15
logical
  See logical address
numbering of for byte locations 3-2
PER
  See PER address
prefixing
  See prefix
primary virtual
  See primary virtual address
real 3-4
secondary virtual
  See secondary virtual address
size of 3-5
  controlled by addressing mode 5-8
storage 3-2
summary information 3-55
translation
  See dynamic address translation, prefix
types 3-3
virtual 3-4
wraparound
  See wraparound
address-limit checking (I/O) 17-23
  effect of I/O-system reset on 17-16
  limit mode (bits in PMCW) 15-2
address-limit-checking control (I/O) 15-24, 16-11
  used for IPL 17-18
address space 3-16
  AR-specified 5-44
  changing of 3-17
control bits
  control bit 5-75
  in PSW 4-6
  use in address translation 3-35
  created by DAT 3-33
number
  See ASN
address-space-control element (ASCE) 3-36
effective 3-42

address-space-control element (ASCE) (continued)
  home 3-38
  in AST entry 3-27
  used for access-register translation 5-57
  used for subspace groups 5-67
primary 3-36
  secondary 3-37
type exception 6-18
address-and-translation-mode identification
  (ATMID) 4-30
addressing exception 6-15
  as an access exception 6-36, 6-43
addressing mode 5-8
  bit in linkage-stack state entry 5-82
  bit in PSW 4-6
  effect on address size 3-6
  effect on operand-address generation 5-9
  effect on sequential instruction-address generation 5-8
  effect on wraparound 3-6
  in branch-address generation 5-10
  in examples A-7
  in operand address generation 5-9
  set by BRANCH AND SAVE AND SET MODE instruction 7-23
  set by BRANCH AND SET MODE instruction 7-24
  set by SET ADDRESSING MODE instruction 7-175
  use of 5-18
ADR (ADD NORMALIZED) HFP instruction 18-8
AE (ADD NORMALIZED) HFP instruction 18-8
  example A-39
AEB (ADD) BFP instruction 19-18
AEFR (ADD) BFP instruction 19-18
AER (ADD NORMALIZED) HFP instruction 18-8
AFP (additional floating-point) registers 9-2
AFP-register data exception 6-21
AFT (ASN first table) 3-25
AFTE (ASN-first-table entry) 3-25
AFTO (ASN-first-table origin) 3-25
AFX (ASN-first-table index) 3-24
  invalid bit 3-25
  translation exception 6-18
AG (ADD) binary instruction 7-18
AGF (ADD) binary instruction 7-18
AGFR (ADD) binary instruction 7-18
AGHI (ADD HALFWORD IMMEDIATE) instruction 7-18
AGR (ADD) binary instruction 7-18
AH (ADD HALFWORD) instruction 7-18
  example A-7
AHI (ADD HALFWORD IMMEDIATE) instruction 7-18
AHY (ADD HALFWORD) instruction 7-18
AKM (authorization key mask) 5-35
AL (ADD LOGICAL) instruction 7-19
ALB (ART-lookaside buffer) 5-62
  entry
  clearing of 5-64
Index X-3

ARL (ART-lookaside buffer) (continued)

entry (continued)
  effect of translation changes on 5-64
  usable state 5-63

ALC (ADD LOGICAL WITH CARRY) instruction 7-20
ALCG (ADD LOGICAL WITH CARRY) instruction 7-20
ALCGR (ADD LOGICAL WITH CARRY) instruction 7-20
ALCR (ADD LOGICAL WITH CARRY) instruction 7-20

ALD (access-list designation) 5-54

ALE
  See access-list entry

ALEAX (access-list-entry authorization index) 5-55
ALEN (access-list-entry number) 5-52
  invalid bit 5-55
  translation exception 6-18
    as an access exception 6-36
  alert (class of machine-check condition) 11-12
  alert interruption condition (I/O) 16-4
  alert-status bit (I/O) 16-16

ALESN (access-list-entry sequence number)
  in ALE 5-55
  in ALET 5-52

ALET (access-list-entry token) 5-46, 5-52
  specification exception 6-18
    as an access exception 6-36

ALG (ADD LOGICAL) instruction 7-19
ALGF (ADD LOGICAL) instruction 7-19
ALGFR (ADD LOGICAL) instruction 7-19
ALGR (ADD LOGICAL) instruction 7-19

alignment 3-3

ALL (access-list length) 5-54

allegiance
  active 15-12
  channel-path 15-11
  dedicated 15-12
  effect on CLEAR SUBCHANNEL of 15-11
  working 15-12

allowed interruptions 6-6

ALO (access-list origin) 5-54

ALR (ADD LOGICAL) instruction 7-19

alter-and-display controls 12-2

alteration
  storage (PER event) 4-35

ALY (ADD LOGICAL) instruction 7-19

ancillary-report bit
  in channel-report word 17-27
  in machine-check-interruption code 11-18
  in subchannel logout 16-34

AND IMMEDIATE instructions 7-21

AND instructions 7-20
  examples A-8

AP (ADD DECIMAL) instruction 8-5
  example A-34

AR (ADD) binary instruction 7-18

AR-specified (access-register-specified) 3-5, 3-16
  address space 3-16, 5-44
  virtual address 3-5

AR-specified (access-register-specified) virtual address
  effective address-space-control element for 3-42

architectural mode
  identification 3-60
  indication of 12-2
  selection of by IML controls 12-3
  selection of by manual controls 12-2
  selection of by signal-processor order 4-58

architecture
  compatibility 1-16

arithmetic
  address
    See address arithmetic
  binary 7-4
    examples A-2
  decimal 8-2
    examples A-4, A-34
  floating-point 9-1
    examples A-5, A-39
  logical (unsigned binary) 7-5
    examples A-3

ART
  See access-register translation

art-lookaside buffer
  See ALB

ASCE
  See address-space-control element

ASCE (address-space-control element)
  type exception 6-18

ASCII character code
  handled by architecture xxi

ASF-control bit
  See address-space-function-control bit

ASN (address-space number) 3-17

authorization 3-30

first table (AFT) 3-25
first-table (AFT) origin (AFTO) 3-25

first-table index
  See AFX

first table origin (AFTO) 3-25
  in entry-table entry 5-35
second table (AST) 3-25
second-table (AST) origin (ASTO) 3-25
second-table address in ETE 5-36
second-table entry (ASTE)
  for subspace groups 5-66
  instance exception 6-19
  instance number (ASTEIN), used in ASN
  reuse 3-19
  origin, in ALE 5-55
  primary (PASTE) 5-32
  pseudo 3-18
  sequence exception 6-19
  sequence exception as an access
  exception 6-36

Index X-3
ASN (address-space number) (continued)
second-table entry (ASTE) (continued)
sequence number (ASTESN), in ALE 5-55
sequence number (ASTESN), in ASTE 5-57
validity exception 6-20
validity exception as an access exception 6-36
second-table index
See ASX
trace-control bit 4-14
translation 3-24
exceptions 6-46
translation-control bit 3-25, 5-26
ASN-and-LX-reuse-control bit 3-19
ASN-and-LX-reuse facility 3-19
assembler language A-7
instruction formats in
See instruction lists and page numbers in
Appendix B
assigned storage locations 3-58
AST (ASN second table) 3-25
AST entry 5-55
See also ASN-second-table entry
ASTE
See ASN-second-table entry
ASTESN (AST-entry sequence number)
in ALE 5-55
in ASTE 5-57
ASTO (ASN-second-table origin) 3-25
ASX (ASN-second-table index) 3-24
invalid bit 3-26
use in ART 5-56
translation exception 6-20
AT
See authority table
ATL (authority-table length) 3-26
use in ART 5-56
ATMID (addressing-and-translation-mode
identification) 4-30
ATO (authority-table origin) 3-26
use in ART 5-56
attached ART-table entry 5-63
attached region-table, segment-table, or page-table
entry 3-50
attachment of I/O devices 13-2
AU (ADD UNNORMALIZED) HFP instruction 18-10
example A-39
AUR (ADD UNNORMALIZED) HFP instruction 18-10
authority table (AT) 5-27
designation 3-26, 5-56
length 3-26, 5-56
origin 3-26, 5-56
authorization
ASN 3-30
index (AX) 3-30, 5-27
key mask (AKM) 5-35
mechanisms 5-25
summary of 5-29
authorization (continued)
testing of 5-73
authorization check 16-37
indicator for (in ERW) 16-37
automatic reconfiguration 1-14
auxiliary storage 3-2, 3-33
availability (characteristic of a system) 1-17
AW (ADD UNNORMALIZED) HFP instruction 18-10
AWR (ADD UNNORMALIZED) HFP instruction 18-10
AX (authorization index) 3-30, 5-27
AXBR (ADD) BFP instruction 19-18
AXR (ADD NORMALIZED) HFP instruction 18-8
AY (ADD) binary instruction 7-18

B
B field of instruction 5-8
backed-up bit (machine-check condition) 11-18
backup
processing (synchronous machine-check
condition) 11-18
backward stack-entry address 5-79
backward stack-entry validity bit 5-79
BAKR (BRANCH AND STACK) instruction 10-10
examples A-10
BAL (BRANCH AND LINK) instruction 7-22
examples A-8
BALR (BRANCH AND LINK) instruction 7-22
examples A-8
BAS (BRANCH AND SAVE) instruction 7-23
example A-8
base address 5-8
register for 2-3
base-AST-entry origin (BASTEO) 5-65
base space 5-15
base-space bit 5-67
base-authority state 10-7
basic addressing mode
bit in entry-table entry 5-35
basic I/O functions 15-1
basic operator facilities 12-1
basic PROGRAM CALL 5-70, 10-66
BASR (BRANCH AND SAVE) instruction 7-23
example A-8
BASSM (BRANCH AND SAVE AND SET MODE)
instruction 7-23
example A-8
BASTEO (base-AST-entry origin) 5-65
BC (BRANCH ON CONDITION) instruction 7-25
example A-11
BCR (BRANCH ON CONDITION) instruction 7-25
BCT (BRANCH ON COUNT) instruction 7-26
example A-12
BCTG (BRANCH ON COUNT) instruction 7-26
BCTGR (BRANCH ON COUNT) instruction 7-26
BCTR (BRANCH ON COUNT) instruction 7-26
  example A-12
BFP
  data class
testing of 19-45
BFP (binary floating point) 9-1
BFP data 19-4
  conversion of 9-8
BFP facility 19-1
BFP-instruction data exception 6-21
bias for exponent 19-4
big endian 7-132
bimodal addressing
  See addressing mode
binary
  See also fixed point
  arithmetic 7-4
    examples A-2
  negative zero 7-3
  number representation 7-3
    examples A-2
  overflow 7-4
    example A-2
  sign bit 7-3
binary floating point (BFP) 9-1
binary integer
  conversion from floating point 18-11, 19-26
  conversion to floating point 18-11, 19-26
binary-to-decimal conversion 7-98
  example A-18
bit 3-2
  numbering of within a group of bytes 3-2
bits to left of compressed data 7-76
bits to right of compressed data 7-76
block-concurrent storage references 5-99
block number
  expanded storage 2-2
  block of I/O data 15-22
  block of storage 3-3
  See also page
testing for usability of 10-127
borrow 7-189
boundary alignment 3-3
  for instructions 5-3
branch address 5-10
  control bit 4-28
  in linkage-stack state entry 5-82
  in trace entry 4-24
BRANCH AND LINK instructions 7-22
  examples A-8
BRANCH AND SAVE AND SET MODE
  instruction 7-23
  examples A-8
BRANCH AND SAVE instructions 7-23
  examples A-8
BRANCH AND SET AUTHORITY instruction 10-6
BRANCH AND SET MODE instruction 7-24
  examples A-8
BRANCH AND STACK instruction 10-10
  examples A-10
BRANCH IN SUBSPACE GROUP instruction 10-13
BRANCH ON CONDITION instructions 7-25
  example A-11
BRANCH ON COUNT instructions 7-26
  example A-12
BRANCH ON INDEX HIGH instruction 7-27
  examples A-12
BRANCH ON INDEX LOW OR EQUAL
  instruction 7-27
  examples A-13
BRANCH RELATIVE AND SAVE instruction 7-28
BRANCH RELATIVE AND SAVE LONG
  instruction 7-28
BRANCH RELATIVE ON CONDITION instruction 7-29
BRANCH RELATIVE ON CONDITION LONG
  instruction 7-29
BRANCH RELATIVE ON COUNT instruction 7-29
BRANCH RELATIVE ON INDEX HIGH
  instruction 7-30
BRANCH RELATIVE ON INDEX LOW OR EQUAL
  instruction 7-30
branch state entry 5-80, 10-10
branch-trace-control bit 4-14
branching
  branch-address generation 5-10
  in a channel program 15-41
  relative 5-10
  to perform decision making, loop control, and sub-
  routine linkage 5-11
  using the linkage stack 5-72
BRAS (BRANCH RELATIVE AND SAVE)
  instruction 7-28
BRASL (BRANCH RELATIVE AND SAVE LONG)
  instruction 7-28
BRC (BRANCH RELATIVE ON CONDITION)
  instruction 7-29
BRCL (BRANCH RELATIVE ON CONDITION LONG)
  instruction 7-29
BRCT (BRANCH RELATIVE ON COUNT)
  instruction 7-29
BRCTG (BRANCH RELATIVE ON COUNT)
  instruction 7-30
broadcasted-purging facility 1-9
BRXH (BRANCH RELATIVE ON INDEX HIGH)
  instruction 7-30
BRXHG (BRANCH RELATIVE ON INDEX HIGH)
  instruction 7-30
BRXLE (BRANCH RELATIVE ON INDEX LOW OR
  EQUAL) instruction 7-30
BRXLG (BRANCH RELATIVE ON INDEX LOW OR
  EQUAL) instruction 7-30
BSA (BRANCH AND SET AUTHORITY) instruction 10-6
BSG (BRANCH IN SUBSPACE GROUP) instruction 10-13
BSM (BRANCH AND SET MODE) instruction 7-24 example A-8
buffer storage (cache) 3-2
burst mode (channel-path operation) 13-3
busy
  in I/O operations 13-7
  in SIGNAL PROCESSOR 4-60
BXH (BRANCH ON INDEX HIGH) instruction 7-27 examples A-12
BXHG (BRANCH ON INDEX HIGH) instruction 7-27 examples A-12
BXLE (BRANCH ON INDEX LOW OR EQUAL) instruction 7-27 examples A-13
BXLEG (BRANCH ON INDEX LOW OR EQUAL) instruction 7-27
bypassing POST and WAIT A-45
byte 3-2
  numbering of in storage 3-2
byte index (BX) 3-34
byte-multiplex mode (channel-path operation) 13-3
CCW (channel-command word) (continued)
  IDA flag in 15-28
  indirect data addressing used in 13-7, 15-36
  invalid format of 16-25
  invalid specification of 16-24
  PCI flag in 15-28
  prefetch control in 16-10
  used for IPL 17-18
  prefetching 15-33
  retry of
    See command retry
    role in I/O operations of 13-6
    skip flag in 15-28
    suspend flag in 15-28
CCW-type IPL (CCW-type initial program loading) 17-18
CD (COMPARE) HFP instruction 18-10
CDB (COMPARE) BFP instruction 19-23
CDBR (COMPARE) BFP instruction 19-23
CDFBR (CONVERT FROM FIXED) BFP instruction 19-26
CDFR (CONVERT FROM FIXED) HFP instruction 18-11
CDGR (CONVERT FROM FIXED) BFP instruction 19-26
CDGBR (CONVERT FROM FIXED) BFP instruction 19-26
CDGR (CONVERT FROM FIXED) HFP instruction 18-11
CDR (COMPARE) HFP instruction 18-10 examples A-40
CDS (COMPARE DOUBLE AND SWAP) instruction 7-53 examples A-44
CDSG (COMPARE DOUBLE AND SWAP) instruction 7-53
CDSS (compressed-data symbol size) 7-74
CDSY (COMPARE DOUBLE AND SWAP) instruction 7-53
CE (COMPARE) HFP instruction 18-10
CEB (COMPARE) BFP instruction 19-23
CEBR (COMPARE) BFP instruction 19-23
CEFBR (CONVERT FROM FIXED) BFP instruction 19-26
CEFGR (CONVERT FROM FIXED) HFP instruction 18-11
CFDBR (CONVERT TO FIXED) BFP instruction 19-26
CCW (channel-command word)
  C (COMPARE) binary instruction 7-46
  cache 3-2
called-space identification 5-82
cancel-I/O facility 1-10
cANCEL SUBCHANNEL instruction 14-4
  capability list 5-49
carry 7-3
CBC (checking-block code) 11-2
  invalid 11-2
    in registers 11-10
    in storage 11-6
    in storage keys 11-7
    near-valid 11-2
    valid 11-2
CBN (compressed-data bit number) 7-73 example of 7-79
CCC (channel-control check) 16-27
CCW (channel-command word) 15-27
  address of 16-18
  byte count in 15-28
  chaining 15-31
  check (in subchannel logout) 16-33
  command codes
    See commands
  contents of 15-27
  current 15-27
  designation of storage area in 15-28, 15-29
  format-0 and format-1 15-27
  format control 16-10
  used for IPL 17-18
C (COMPARE) binary instruction 7-46
  cache 3-2
called-space identification 5-82
cancel-I/O facility 1-10
cANCEL SUBCHANNEL instruction 14-4
  capability list 5-49
carry 7-3
CBC (checking-block code) 11-2
  invalid 11-2
    in registers 11-10
    in storage 11-6
    in storage keys 11-7
    near-valid 11-2
    valid 11-2
CBN (compressed-data bit number) 7-73 example of 7-79
CCC (channel-control check) 16-27
CCW (channel-command word) 15-27
  address of 16-18
  byte count in 15-28
  chaining 15-31
  check (in subchannel logout) 16-33
  command codes
    See commands
  contents of 15-27
  current 15-27
  designation of storage area in 15-28, 15-29
  format-0 and format-1 15-27
  format control 16-10
  used for IPL 17-18
Index

CFDR (CONVERT TO FIXED) HFP instruction 18-11
CFEVR (CONVERT TO FIXED) BFP instruction 19-26
CFER (CONVERT TO FIXED) HFP instruction 18-11
CFXBR (CONVERT TO FIXED) BFP instruction 19-26
CFXR (CONVERT TO FIXED) HFP instruction 18-11
CG (COMPARE) binary instruction 7-46
CGDBR (CONVERT TO FIXED) BFP instruction 19-26
CGDR (CONVERT TO FIXED) HFP instruction 18-11
CGEBR (CONVERT TO FIXED) BFP instruction 19-26
CGER (CONVERT TO FIXED) HFP instruction 18-11
CGF (COMPARE) binary instruction 7-46
CGFR (COMPARE) binary instruction 7-45
CGHI (COMPARE HALFWORD IMMEDIATE) instruction 7-55
CGR (COMPARE) binary instruction 7-45
CGXBR (CONVERT TO FIXED) BFP instruction 19-26
CGXR (CONVERT TO FIXED) HFP instruction 18-11
CH (COMPARE HALFWORD) instruction 7-55
example A-14
chaining check (subchannel status) 16-29
chaining of CCWs 15-31
command
See command chaining of CCWs
data
See data chaining of CCWs
chaining of CRWs 17-26, 17-27
change bit in storage key 3-8
change recording 3-14
channel-command word
See CCW
channel commands
See commands (I/O)
channel-control check (subchannel status) 16-27
channel-data check (subchannel status) 16-26
channel path 13-2
active allegiance for 15-12
available for selection 15-13
dedicated allegiance for 15-12
effect of I/O-system reset on 17-15
masks in SCHIB
See LPM, LPUM, PAM, PIM, PNOM, POM
multipath mode of 15-3, 15-21
not operational 16-12
parallel-I/O-interface type 13-3
serial-I/O-interface type 13-2
storing of status for 14-16
type of 13-2, 13-5
working allegiance for 15-12
channel-path identifier
See CHPID
channel-path reset 17-13
effect of I/O-system reset on 17-15
channel-path-reset function 15-45
completion of 15-45
initiation by RESET CHANNEL PATH 14-9
reset signal issued as part of 17-13
channel-path-reset function (continued)
signaling for 15-45
channel-path-status word 14-16
channel-path timeout
indicator for (in ERW) 16-37
channel program 15-27
2K-IDAW control for 15-25
branching in
See TIC
execution of 13-6, 15-21
resumption of 14-11
suspension of 13-8, 15-38
format-2-IDAW control for 15-25
modification control for 15-23
serialization 5-103
streaming-mode control for 15-23
suspend control for 15-22
synchronization control for 15-24
channel-program address 16-18
field-validity flag for in IRB 16-34
used for IPL 17-18
channel report 17-25
generated as a result of RCHP 14-9
channel report pending 11-17, 17-25
effect of I/O-system reset on 17-16
subclass-mask bit for 11-24
channel-report word
See CRW
channel subsystem 2-6, 13-1
addressing used in 13-5
damage 11-18
effect of I/O-system reset on 17-14
effect of power-on reset on 4-54
isolated state of 16-37
channel-subsystem-call facility 1-13
channel-subsystem monitoring 17-1
effect of I/O-system reset on 17-16
channel-subsystem recovery 11-4, 17-24
channel-subsystem timer 17-2
effect of I/O-system reset on 17-16
channel-subsystem timing 17-2
channel-subsystem timing-facility bit (in PMCW) 15-4
characteristic (of HFP number) 18-1
See also exponent
characters
represented by eight-bit code xxi
check bits 3-2, 11-2
check stop 4-3, 11-11
as signal-processor status 4-62
during manual operation 12-1
effect on CPU timer 4-46
entering of 11-13
indicator 12-3
malfunction alert for 6-12
system 11-11
checking block 11-2
checking-block code
   See CBC
checkpoint 11-2
checkpoint synchronization 11-3
   action 11-4
   operations 11-3
CHECKSUM instruction 7-31
CHI (COMPARE HALFWORD IMMEDIATE) instruction 7-55
CHPID (channel-path identifier) 13-5
   in PMCW 15-7
   used in RESET CHANNEL PATH 14-9
CHY (COMPARE HALFWORD) instruction 7-55
CIPHER MESSAGE instruction 7-35
CIPHER MESSAGE WITH CHAINING instruction 7-35
CKSM (CHECKSUM) instruction 7-31
CL (COMPARE LOGICAL) instruction 7-56
class of BFP data 19-5
   testing of 19-45
CLC (COMPARE LOGICAL) instruction 7-56
   example A-14
CLCL (COMPARE LOGICAL LONG) instruction 7-58
   example A-16
CLCLE (COMPARE LOGICAL LONG EXTENDED) instruction 7-60
CLCLU (COMPARE LOGICAL LONG UNICODE) instruction 7-64
clear function 15-14
   bit in SCSW for 16-13
   completion of 15-15
   initiated by CLEAR SUBCHANNEL 14-5
   path management for 15-14
   pending 16-15
   signaling for 15-15
   subchannel modification by 15-14
clear reset 4-53
clear signal 17-12
   issued as part of clear function 15-15
CLEAR SUBCHANNEL instruction 14-5
   See also clear function
   effect on device status of 15-15
   function initiated by 15-14
   use of after RESET CHANNEL PATH 14-10
clearing operation
   by clear-reset function 4-53
   by load-clear key 12-3
   by load-clear-list-directed key 12-3
   by system-reset-clear key 12-5
   by TEST BLOCK instruction 10-127
CLG (COMPARE LOGICAL) binary instruction 7-56
CLGF (COMPARE LOGICAL) binary instruction 7-56
CLGFR (COMPARE LOGICAL) binary instruction 7-56
CLGR (COMPARE LOGICAL) binary instruction 7-56
CLI (COMPARE LOGICAL) instruction 7-56
   example A-15
CLIY (COMPARE LOGICAL) instruction 7-56
CLM (COMPARE LOGICAL CHARACTERS UNDER MASK) instruction 7-57
   example A-15
CLMH (COMPARE LOGICAL CHARACTERS UNDER MASK) instruction 7-57
CLMY (COMPARE LOGICAL CHARACTERS UNDER MASK) instruction 7-57
clock
   See TOD clock
clock comparator 4-45
   external interruption 6-11
   save areas for 3-63
   validity bit for 11-22
clock unit 4-44, 7-185
CLR (COMPARE LOGICAL) instruction 7-56
   example A-15
CLST (COMPARE LOGICAL STRING) instruction 7-67
   examples A-17
CLY (COMPARE LOGICAL) instruction 7-56
CMPSC (COMPRESSION CALL) instruction 7-72
code
   ASCII
   handled by architecture xxi
   checking-block
   See CBC
   command (in CCW)
   See command code in CCW
   condition
   See condition code
data-exception (DXC) 6-14
decimal digit and sign 8-2
defered condition (I/O) 16-8
EBCDIC
   handled by architecture xxi
table for I-1
eight-bit
   handled by architecture xxi
error-recovery (I/O) 17-27
external-damage 11-23
   validity bit for 11-21
I/O-interruption subclass 15-2
instruction-length
   See ILC
   interruption
   See interruption code
   linkage-stack-entry type 5-78
   monitor
   See monitor code
   operation 5-2
   PER
   See PER code
   reporting-source (I/O) 17-27
   storage-access (in subchannel logout) 16-34
code (continued)  
version 10-108  
codeword (for sorting operations) 7-46  
example A-52  
command chaining of CCWs 15-34  
effect of status modifier on 15-35  
flag in CCW for 15-28  
overview of 13-8  
command code in CCW 15-29  
See also commands  
See also common I/O-device commands  
applicable flags 15-40  
invalid 16-24  
command codes  
See command code in CCW  
command retry 15-42  
effect on PCI of 15-36  
commands (I/O) 15-29  
See also common I/O-device commands  
transfer in channel 15-41  
common I/O-device commands  
publishation referenced xxi  
common-segment bit 3-40  
COMPARE AND FORM CODEWORD instruction 7-46  
example A-51  
COMPARE AND SIGNAL BFP instructions 19-24  
COMPARE AND SWAP instructions 7-53  
COMPARE AND SWAP AND PURGE  
instruction 10-18  
COMPARE AND SWAP instruction  
examples A-44  
COMPARE BFP instructions 19-23  
COMPARE binary instructions 7-45  
COMPARE DECIMAL instruction 8-6  
example A-34  
COMPARE DOUBLE AND SWAP instruction  
examples A-44  
COMPARE DOUBLE AND SWAP instructions 7-53  
COMPARE HALFWORD IMMEDIATE  
instructions 7-55  
COMPARE HALFWORD instruction  
example A-14  
COMPARE HALFWORD instructions 7-55  
COMPARE HFP instructions 18-10  
examples A-40  
COMPARE LOGICAL instructions 7-56  
COMPARE LOGICAL CHARACTERS UNDER MASK  
instruction  
example A-15  
COMPARE LOGICAL CHARACTERS UNDER MASK  
instructions 7-57  
COMPARE LOGICAL instructions  
examples A-14  
COMPARE LOGICAL LONG EXTENDED  
instruction 7-60  
COMPARE LOGICAL LONG instruction 7-58  
example A-16  
COMPARE LOGICAL LOGICAL LONG UNICODE  
instruction 7-64  
COMPARE LOGICAL STRING instruction 7-67  
examples A-17  
COMPARE UNTIL SUBSTRING EQUAL  
instruction 7-68  
comparison  
address  
See address comparison  
decimal 8-6  
example A-34  
hexadecimal-floating-point  
examples A-40  
logical 7-5  
examples A-14  
of BFP data 19-8  
signed-binary 7-5  
TOD-clock 4-45  
compatibility 1-16  
among systems implementing different  
architectures 1-17  
among systems implementing same  
architecture 1-16  
control-program 1-17  
problem-state 1-17  
completion of I/O functions  
by channel-path-reset function 15-45  
by clear function 15-15  
by halt function 15-16  
during data transfer 15-43  
during initiation 15-42  
for immediate commands 15-43  
start and resume 15-42  
completion of instruction execution 5-20  
completion of unit of operation 5-22  
compressed-data bit number (CBN) 7-73  
examples 7-79  
compressed-data symbol size (CDSS) 7-74  
COMPRESSION CALL instruction 7-72  
restriction related to  
on leftmost bits in symbol-translation-table  
entry 7-77  
on length of chain of compression-dictionary  
entries 7-78  
on length of chain of expansion-dictionary  
entries 7-78  
on length of character symbol during  
compression 7-78  
on length of character symbol during  
expansion 7-78  
on number of children 7-78, 7-80  
on offset to symbol-translation table 7-75  
on symbol translation with format-1 sibling  
descriptors 7-76
compression dictionary
    size of with symbol translation  7-75
compression facility
    publication referenced  xxi
compression process
    flowchart of  7-81
COMPUTE INTERMEDIATE MESSAGE DIGEST
    instruction  7-84
COMPUTE LAST MESSAGE DIGEST
    instruction  7-84
COMPUTE MESSAGE AUTHENTICATION CODE
    instruction  7-91
conceptual sequence  5-89
    as related to storage-operand accesses  5-101
conclusion of I/O operations  13-8, 16-1
    during data transfer  15-43
    during initiation  15-42
    for immediate commands  15-43
    prior to initiation  15-42
conclusion of instruction execution  5-20
concurrency of access for storage references  5-99
concurrent sense
    in ECW  16-40
    indicator for (in ERW)  16-37
concurrent-sense count (in ERW)  16-37
concurrent-sense facility  17-24
condition code  4-6
    deferred  16-8
    for BFP instructions  19-9
    in PSW  4-6
summary  C-1
    tested by BRANCH ON CONDITION
        instruction  7-25
    used for decision making  5-11
    validity bit for  11-21
conditional-swapping instructions
    See COMPARE AND SWAP instruction, COMPARE
        DOUBLE AND SWAP instruction
conditions for interruption
    See interruption conditions
collection of storage  3-3
collection-alert facility (I/O)  17-24
connective
    See logical connective
consistency (storage operand)  5-98
    examples  A-47, A-49
console device  12-1
console integration  1-14
control  4-1
    instructions  10-1
    manual
        See manual operation
control-program compatibility  1-17
control register  2-4, 4-7
    save areas for  3-64
    validity bit  11-21
control-register assignment  4-8
    (CRx.y indicates control register x, bit position y)
CR0.33:
    SSM-suppression-control bit  6-31, 10-105
CR0.34:
    TOD-clock-sync-control bit  4-40, 4-44
CR0.35:
    low-address-protection-control bit  3-12
CR0.36:
    extraction-authority-control bit  5-25
CR0.37:
    secondary-space-control bit  3-36, 5-26
CR0.38:
    fetch-protection-override-control bit  3-11
CR0.39:
    storage-protection-override-control bit  3-10
CR0.44:
    ASN-and-LX-reuse-control bit  3-19
CR0.45:
    AFP-register-control bit  9-2
CR0.48:
    malfunction-alert subclass-mask bit  6-12
CR0.49:
    emergency-signal subclass-mask bit  6-11
CR0.50:
    external-call subclass-mask bit  6-12
CR0.52:
    clock-comparator subclass-mask bit  6-11
CR0.53:
    CPU-timer subclass-mask bit  6-11
CR0.54:
    service-signal subclass-mask bit  6-13
CR0.57:
    interrupt-key subclass-mask bit  6-12
CR0.59:
    ETR subclass-mask bit  6-12
CR0.61:
    crypto-control bit  6-20
CR1:
    primary address-space-control element
        (PASCE)  3-36
CR1.0-51:
    primary real-space token origin (PRSKTO)  3-37
CR1.0-51:
    primary region-table origin (PRTO)  3-36
    primary segment-table origin (PSTO)  3-36
CR1.54:
    primary subspace-group-control bit  3-36
CR1.55:
    primary private-space-control bit  3-36
CR1.56:
    primary storage-alteration-event-control bit  3-36
CR1.57:
    primary space-switch-event-control bit  3-36, 6-30
CR1.58:
    primary real-space-control bit  3-37
CR1.60-61: 
    primary designation-type-control bits 3-37
CR1.62-63: 
    primary region-table length (PRTL) 3-37
    primary segment-table length (PSTL) 3-37
CR2.33-57: 
    dispatchable-unit-control-table origin (DUCTO) 5-51
CR3.0-31: 
    secondary ASTEIN (SASTEIN) 3-19
CR3.32-47: 
    PSW-key mask (PKM) 5-26
CR3.48-63: 
    secondary ASN (SASN) 3-17
CR4.0-31: 
    primary ASTEIN (PASTEIN) 3-19
CR4.32-47: 
    authorization index (AX) 3-30, 5-27
CR4.48-63: 
    primary ASN (PASN) 3-17
CR5.33-57: 
    primary-AST-entry origin (PASTEO) 5-32, 5-52
CR6.32-39: 
    I/O-interruption subclass mask 6-13
CR7: 
    secondary address-space-control element (SASCE) 3-37
CR7.0-51: 
    secondary real-space token origin (SRSTKO) 3-37
    secondary region-table origin (SRTO) 3-37
    secondary segment-table origin (SSTO) 3-37
CR7.54: 
    secondary subspace-group-control bit 3-37
CR7.55: 
    secondary private-space-control bit 3-37
CR7.56: 
    secondary storage-alteration-event-control bit 3-37
CR7.58: 
    secondary real-space-control bit 3-37
CR7.60-61: 
    secondary designation-type-control bits 3-37
CR7.62-63: 
    secondary region-table length (SRTL) 3-37
    secondary segment-table length (SSTL) 3-37
CR8.32-47: 
    extended authorization index (EAX) 5-52
CR8.48-63: 
    monitor-mask bits 6-25
CR9.36: 
    PER store-using-real-address-event-mask bit 4-28
CR9.40: 
    PER branch-address-control bit 4-28
control-register assignment (continued)

CR14.45-63:
   ASN-first-table origin (AFTO) 3-25
CR15.0-60:
   linkage-stack-entry address 5-77
control unit 2-7, 13-4
   effect of I/O-system reset on 17-14
   sharing of 13-4
   type of 15-13
control unit defer time (I/O) 17-9
control unit defer time interval (in measurement block) 17-7
control-unit-queueing measurement (I/O) 17-9
control-unit-queueing-time interval (in measurement block) 17-6
controlled-ASN bit 3-19
conversion
   between HFP and BFP data 9-8
   binary-to-decimal 7-98
      example A-18
   decimal-to-binary 7-97
      example A-18
   hexadecimal-floating-point-number
      basic example A-6
      examples with instructions A-42
   of floating-point format 19-7
CONVERT BFP TO HFP floating-point instructions 9-8
CONVERT FROM FIXED BFP instructions 19-26
CONVERT FROM FIXED HFP instructions 18-11
CONVERT HFP TO BFP floating-point instructions 9-9
CONVERT TO BINARY instruction
   example A-18
CONVERT TO BINARY instructions 7-97
CONVERT TO DECIMAL instruction
   example A-18
CONVERT TO DECIMAL instructions 7-98
CONVERT TO FIXED BFP instructions 19-26
CONVERT TO FIXED HFP instructions 18-11
CONVERT UNICODE TO UTF-8 instruction 7-101
CONVERT UTF-16 TO UTF-32 instruction 7-98
CONVERT UTF-16 TO UTF-8 instruction 7-101
CONVERT UTF-32 TO UTF-16 instruction 7-104
CONVERT UTF-32 TO UTF-8 instruction 7-107
CONVERT UTF-8 TO UNICODE instruction 7-110
CONVERT UTF-8 TO UTF-16 instruction 7-110
CONVERT UTF-8 TO UTF-32 instruction 7-113
Coordinated Universal Time (UTC) used in TOD epoch 4-42
COPY ACCESS instruction 7-116
   count field
      in CCW 15-28
      invalid 16-24
      in SCSW 16-29
counter updating (example) A-45
counting operations 7-26
coupling facility 1-14
CP (COMPARE DECIMAL) instruction 8-6
   example A-34
CPA
   See channel-program address
CPU (central processing unit) 2-2
   address 4-55
      assigned storage locations for 3-58
      when stored during external interruptions 6-10
   checkpoint 11-2
   effect of power-on reset on 4-53
   hangup due to string of interruptions 4-3
   identification (ID) 10-108
   machine-type number 10-108
   model number 10-108
   registers 2-3
      save areas for 3-63
   reset 4-51
      signal-processor order 4-57
   retry 11-2
   serialization 5-102
   signaling 4-56
   state 4-1
      check-stop 4-3
      load 4-2
      no effect on TOD clock 4-40
      operating 4-2
      stopped 4-2
      version code 10-108
CPU timer 4-46
   external interruption 6-11
   save areas for 3-63
   validity bit for 11-22
CPYA (COPY ACCESS) instruction 7-116
CR
   See control register
CR (COMPARE) binary instruction 7-45
CRW (channel-report word) 17-26
   chaining of 17-26, 17-27
   error-recovery code (ERC) in 17-27
   overflow in 17-26
   reporting-source code (RSC) in 17-27
   reporting-source ID (RSID) in 17-28
   solicited 17-26
   storing of 14-17
crypto-operation exception 6-20
cryptographic facility 1-13, 2-6
CS (COMPARE AND SWAP) instruction 7-53
   examples A-44
CSCH (CLEAR SUBCHANNEL) instruction 14-5
CSG (COMPARE AND SWAP) instruction 7-53
CSP (COMPARE AND SWAP AND PURGE) instruction 10-18
CSPG (COMPARE AND SWAP AND PURGE) instruction 10-18
CSY (COMPARE AND SWAP) instruction 7-53
CU12 (CONVERT UTF-8 TO UTF-16) instruction 7-110
CU14 (CONVERT UTF-8 TO UTF-32) instruction 7-113
CU21 (CONVERT UTF-16 TO UTF-8) instruction 7-107
CU41 (CONVERT UTF-32 TO UTF-8) instruction 7-107
CU42 (CONVERT UTF-32 TO UTF-16) instruction 7-104

current CCW 15-27
   See also CCW

current PSW 4-3, 5-11
   See also PSW
   stored during interruption 6-2

CUSE (COMPARE UNTIL SUBSTRING EQUAL) instruction 7-68
CUTFU (CONVERT UTF-8 TO UNICODE) instruction 7-110
CUUTF (CONVERT UNICODE TO UTF-8) instruction 7-101

CVB (CONVERT UNICODE TO UTF-8) instruction 7-97
example A-18
CVBG (CONVERT TO BINARY) instruction 7-97
CVBY (CONVERT TO BINARY) instruction 7-97
CVD (CONVERT TO DECIMAL) instruction 7-98
example A-18
CVDG (CONVERT TO DECIMAL) instruction 7-98
CVDY (CONVERT TO DECIMAL) instruction 7-98
CXBR (COMPARE) BFP instruction 19-23
CXFR (CONVERT FROM FIXED) BFP instruction 19-26
CXGBR (COMPARE) BFP instruction 19-23
CXGR (CONVERT FROM FIXED) BFP instruction 19-23
CXR (COMPARE) HFP instruction 18-10
CY (COMPARE) binary instruction 7-46

damage (continued)
   processing 11-19
   service-processor 11-18
   system 11-16
   timing-facility 11-16

DAT
   See dynamic address translation
DAT mode (bit in PSW) 4-5
   use in address translation 3-35
DAT-table format error 6-36

data
   blocking of (I/O) 15-22
   format for
      binary-floating-point instructions 19-4
      decimal instructions 8-1
      general instructions 7-2
      hexadecimal-floating-point instructions 18-3
   indirect addressing of (I/O) 13-7, 15-36
   measurement (I/O)
      See measurement data
      prefetching of for I/O operation 15-30
   data address (I/O) 15-29
   invalid 16-25
   invalid specification of 16-25
   data chaining of CCWs 15-33
      flag in CCW for 15-28
      overview of 13-8
   data check
      measurement-block 16-33
   data exception 6-20
      AFP-register 6-21
      BFP-instruction 6-21
      decimal-operand 6-21, 8-4
      IEEE-exception-condition 6-21, 19-10
      priority of program interruptions for 6-14
   data exception code
      See DXC
   data exception code (DXC) 6-14
   data streaming (I/O) 13-3
      effect of CCW count on 15-34
   DCTI (device-connect-time interval)
      in ESW 16-39
      in extended-measurement word 16-42
      in measurement block 17-5

DD (DIVIDE) HFP instruction 18-12
DDB (DIVIDE) BFP instruction 19-29
DDBR (DIVIDE) BFP instruction 19-29
DDR (DIVIDE) HFP instruction 18-12
DE (DIVIDE) HFP instruction 18-12
DEB (DIVIDE) BFP instruction 19-29
DEBR (DIVIDE) BFP instruction 19-29

decimal
   arithmetic 8-2
   comparison 8-6
   digit codes 8-2
   divide exception 6-21

D
D (DIVIDE) binary instruction 7-116
example A-19
D field of instruction 5-9

damage
   channel-subsystem 11-18
   code (external) 11-23
      validity bit for 11-21
   external 11-17
      subclass-mask bit for 11-25
   instruction-processing 11-16

Index  

X-13
decimal (continued)

instructions  8-1
  examples A-34
number representation  8-1
  examples A-4
operand overlap  8-3
overflow
  exception  6-21
  mask in PSW  4-6
  sign codes  8-2
decimal-operand data exception  6-21, 8-4
decimal-to-binary conversion  7-97
  example A-18
dedicated allegiance  15-12
default QNaN  19-8
defered condition code  16-8
degradation (machine-check condition)  11-17
  subclass-mask bit for  11-25
degradation, storage (machine-check condition)  11-20
delay in storing  5-96
delayed access exception (machine-check condition)  11-18
deletion of malfunctioning unit  11-4
denormalized numbers  19-8
DER (DIVIDE) HFP instruction  18-12
  examples A-40
designation
  authority-table  3-26
  entry-table  5-33, 5-34
  home real-space  3-38
  home region-table  3-38
  home segment-table  3-38
  linkage-first-table  5-33
  linkage-table  5-32
  of storage area for data (I/O)  15-29
  page-table  3-40
  primary real-space  3-36
  primary region-table  3-36
  primary segment-table  3-36
  real-space  3-36
    in AST entry  3-27
    used for access-register translation  5-57
    used for subspace groups  5-67
  region-table  3-36
    in AST entry  3-27
    used for access-register translation  5-57
    used for subspace groups  5-67
  secondary real-space  3-37
  secondary region-table  3-37
  secondary segment-table  3-37
  segment-table  3-36
    in AST entry  3-27
    used for access-register translation  5-57
    used for subspace groups  5-67
designation (origin and length)
  access-list  5-53
designation-type-control bits
  home  3-38
  primary  3-37
  secondary  3-37
destructive overlap  5-100, 7-135, 7-139, 7-144
  in the access-register mode  5-91
device  2-7, 13-4
  console  12-1
    effect of I/O-system reset on  17-14
device-active bit  16-15
device-active-only measurement (I/O)  17-9
device-active-only-time interval (in measurement block)  17-6
device address  13-5
device-busy time (in extended measurement word)  16-42
device-busy time (in measurement block)  17-6
device-connect-time interval
  See DCTI
device-connect-time measurement  17-10
  effect of suspension on  15-40
  enable  15-3
device-disconnect-time interval (in measurement block)  17-5
device identifier  13-5
device number  13-5
  assignment of  13-5
    in PMCW  15-4
device-number valid (bit in PMCW)  15-4
device status  16-23
    field-validity flag for (in subchannel logout)  16-28,
      16-34
  with inappropriate bit combination  16-35
device status check  16-35
  DIAGNOSE instruction  10-19
dictionary
  adaptive  7-80
  DIDBR (DIVIDE TO INTEGER) BFP instruction  19-29
  DIEBR (DIVIDE TO INTEGER) BFP instruction  19-29
digit codes (decimal)  8-2
digit selector (in EDIT)  8-7
direct-access storage  3-2
disabling for interruptions  6-6
disallowed interruptions  6-6
dispatchable unit (DU)  5-46
  access-list designation (DUALD)  5-53
  control table (DUCT)  5-53
  origin (DUCTO)  5-51
    when subspace-group facility installed  5-64
    when trap facility installed  10-133
displacement (in relative addressing)  5-9
display (manual controls)  12-2
  DIVIDE BFP instructions  19-29
  DIVIDE binary instructions  7-116
  example A-19
DIVIDE DECIMAL instruction 8-6
  example A-34
divide exception
decimal 6-21
  fixed-point 6-22
  HFP 6-23
DIVIDE HFP instructions 18-12
  examples A-40
DIVIDE LOGICAL instructions 7-117
DIVIDE SINGLE binary instructions 7-117
DIVIDE TO INTEGER BFP instructions 19-29
  remainder result of 19-9
divisible instruction execution 5-89
DL (DIVIDE LOGICAL) instruction 7-117
DLG (DIVIDE LOGICAL) binary instruction 7-117
DLGR (DIVIDE LOGICAL) binary instruction 7-117
DLR (DIVIDE LOGICAL) instruction 7-117
doubleword 3-3
doubleword-concurrent storage references 5-99
DP (DIVIDE DECIMAL) instruction 8-6
  example A-34
DR (DIVIDE) binary instruction 7-116
DSG (DIVIDE SINGLE) binary instruction 7-117
DSGFR (DIVIDE SINGLE) binary instruction 7-117
DSGR (DIVIDE SINGLE) binary instruction 7-117
DU (dispatchable unit) 5-46
DUALD (dispatchable-unit access-list
  designation) 5-53
DUCT (dispatchable-unit control table) 5-53, 5-64
DUCTO (dispatchable-unit-control-table origin) 5-51
dump (standalone) 12-5
DXBR (DIVIDE) BFP instruction 19-29
DXC (data-exception code) 6-14, 19-14
  summary figure 19-14
  summary figure for IEEE 19-14
DXR (DIVIDE) HFP instruction 18-12
dynamic address translation (DAT) 3-33
  by LOAD REAL ADDRESS instruction 10-47
  by STORE REAL ADDRESS instruction 10-111
  control of 3-35
  explicit and implicit 3-41
  mode bit in PSW 4-5
  use in address translation 3-35
  sequence of table fetches 5-94
dynamic-reconnection feature 13-2

E
E (exa) xx
E instruction format 5-5
EAR (EXTRACT ACCESS) instruction 7-120
  early exception recognition 6-9
EAX
  See extended authorization index
EBCDIC (Extended Binary-Coded-Decimal Interchange
  Code)
  architecture designed for xx
  character code
  table for I-1
ECC (error checking and correction) 11-2
ECW (extended-control word) 16-40
  indication in SCSW 16-11
ED (EDIT) instruction 8-7
  examples A-35
EDIT AND MARK instruction 8-9
  example A-36
EDIT instruction 8-7
  examples A-35
editing instructions 8-3
  See also ED instruction, EDMK instruction
EDMK (EDIT AND MARK) instruction 8-9
  example A-36
effective access-list designation 5-53
effective address 3-5
controlled by addressing mode 5-8
generation 5-8
  used for storage interlocks 5-90
effective address-space-control element 3-42
EFPC (EXTRACT FPC) instruction 19-33
EKM (entry key mask) 5-35
emergency signal (external interruption) 6-11
  signal-processor order 4-56
EMIF (ESCON-multiple-image facility) 1-14
EMW (extended-measurement word) 16-41
  in IRB 16-41
  enabled (bit for TRAP) 10-134
  enabled (bit in PMCW) 15-2
  enabling for interruptions 6-6
  subchannel 16-5
  enabling of subchannel 15-2, 16-5
  endian 7-132
  ending of instruction execution 5-20
Enterprise Systems Connection Architecture (ESCON)
  I/O interface
  publication referenced xx
  entry
  extended authorization index 5-75
  key 5-75
  entry (for tracing) 4-14
  entry descriptor 5-78
  entry index (EX) 5-31
  entry key mask (EKM) 5-35
  entry table (ET)
    designation 5-33, 5-34
    length (ETL) 5-33, 5-34
    origin (ETO) 5-33, 5-34
  entry table entry 5-34
  entry-table entry (ETE) 5-74
  entry-type code 5-78
EPAIR (EXTRACT PRIMARY ASN AND INSTANCE) instruction 10-21
EPAR (EXTRACT PRIMARY ASN) instruction 10-21
epoch (for TOD clock) 4-42
EPSW (EXTRACT PSW) instruction 7-120
equipment check
   in signal-processor status 4-61
ERG (error-recovery code) 17-27
   See also CRW
EREG (EXTRACT STACKED REGISTERS) instruction 10-22
EREGG (EXTRACT STACKED REGISTERS) instruction 10-22
error
   checking and correction 11-2
   from DIAGNOSE instruction 10-20
I/O-error alert 16-35
indirect storage 11-20
intermittent 11-5
PSW-format 6-9
secondary (I/O) 16-35
solid 11-5
state of TOD clock 4-41
storage 11-19
storage-key 11-20
error-recovery code (ERC) 17-27
   See also CRW
ERW (extended-report word) 16-32, 16-36
   as result of channel-control check 16-27
   as result of channel-data check 16-27
ESA/370 architecture 1-14
ESA/390
   compatibility with z/Architecture 1-17
ESA/390 architecture
   architectural-mode controls 12-2
ESAIR (EXTRACT SECONDARY ASN AND INSTANCE) instruction 10-22
ESAR (EXTRACT SECONDARY ASN) instruction 10-22
ESCON (Enterprise Systems Connection Architecture) 13-2
ESCON (Enterprise Systems Connection Architecture)
   I/O interface
   publication referenced xxi
ESCON channel-to-channel adapter
   publication referenced xxi
ESCON-multiple-image facility (EMIF) 1-14
ESEA (EXTRACT AND SET EXTENDED AUTHORITY) instruction 10-21
ESTA (EXTRACT STACKED STATE) instruction 10-24
ESW (extended-status word) 16-32
   See also extended status
ESW format bit (in SCSW) 16-8
ET
   See entry table
ETE
   See entry-table entry
ETL (entry-table length) 5-33, 5-34
ETO (entry-table origin) 5-33, 5-34
ETR
   external interruption 6-12
ETR (external time reference) 2-6
ETR (external time reference) facility 1-13
ETR sync check (machine-check condition) 11-23
event 6-14
   monitor 7-133
   PER 4-27
   space-switch 6-30
EX (entry index) 5-31
   translation exception 6-21
EX (EXECUTE)
   See EXECUTE instruction
exception access identification 3-59
exceptions 6-14
   access (collective program-interruption name) 6-36,
   6-43
   addressing 6-15
   AFX-translation 6-18
   ALE-sequence 6-18
   ALEN-transl ation 6-18
   ALET-specification 6-18
   ASCE-type 6-18
   ASN-transl ation (collective program-interruption
   name) 6-46
   associated with
   ART 5-62
   stacking process 5-86
   unstacking process 5-88
   ASTE-instance 6-19
   ASTE-sequence 6-19
   ASTE-validity 6-20
   ASX-translation 6-20
   crypto-operation 6-20
   data 6-20
   decimal-divide 6-21
   decimal-overflow 6-21
   delayed access (machine-check condition) 11-18
   during translation 3-49
   EX-translation 6-21
   execute 6-21
   extended-authority 6-22
   fixed-point-divide 6-22
   fixed-point-overflow 6-22
   HFP-divide 6-23
   HFP-exponent-overflow 6-23
   HFP-exponent-underflow 6-23
   HFP-significance 6-23
   HFP-square-root 6-23
   IEEE 19-10
   LFX-translation 6-23
   LSTE-sequence 6-23
exceptions (continued)

LSX-translation 6-24
LX-translation 6-24
operand (of I/O instruction) 6-25
operation 6-26
page-translation 6-26
PC-translation-specification 6-26
primary-authority 6-27
privileged-operation 6-27
protection 6-28
PSW-related 6-9
recognition of early and late 6-9
region-first-translation 6-28
region-second-translation 6-29
region-third-translation 6-29
region translation 6-28
secondary-authority 6-30
segment-translation 6-30
special-operation 6-31
specification 6-33
stack-empty 6-34
stack-full 6-35
stack-operation 6-35
stack-specification 6-35
stack-type 6-35
subspace-replacement (collective program-interruption name) 6-47
trace (collective program-interruption name) 6-47
trace-table 6-35
translation-specification 6-36
EXCLUSIVE OR instructions 7-118
examples A-19
execute exception 6-21
EXECUTE instruction 7-119
effect of address comparison on 12-1
example A-21
exceptions while fetching target of 6-8
PER event for target of 4-34
exigent machine-check conditions 11-11
expanded storage 2-2
block number 2-2
expansion dictionary
location of during compression 7-76
expansion process
flowchart of 7-83
expansion-operation bit 7-73
explicit address translation 3-41
explicit-trace-control bit 4-14
exponent 18-1
See also characteristic, floating point
See also floating point
overflow
HFP 18-1
underflow
HFP 18-1
mask in PSW 4-6

exponent bias 19-4
extended addressing mode
bit in entry-table entry 5-35
bit in trace entry 4-25
extended-authority exception 6-22
as an access exception 6-36
extended authorization 5-61
extended authorization index (EAX) 5-52
control bit 5-74
in entry-table entry 5-75
in linkage-stack state entry 5-81
extended binary-floating-point number 19-4
extended control (bit in SCSW) 16-11
extended-control word 16-40
See also ECW
extended hexadecimal-floating-point number 18-3
extended-measurement-word 17-11
extended measurement word (I/O)
update enable 15-8
extended-measurement-word enable (I/O)
extended measurement word mode enable (I/O) 15-8
extended-report word
See ERW
extended-sorting facility 1-13
extended status
See also ESW
flags in subchannel logout for 16-32
format-0 16-32
format-1 16-38
format-2 16-39
format-3 16-40
secondary-CCW address 16-38
extended-status word 16-32
See also extended status
extended-status-word-format bit 16-8
extended-subchannel-logout descriptor (in ERW) 16-38
in ESW 16-38
extended-subchannel-logout pending 16-36
indicator for (in ERW) 16-36
external call
external interruption 6-12
pending (signal-processor status) 4-62
signal-processor order 4-56
external damage 11-17
subclass-mask bit for 11-25
external-damage code 11-23
assigned storage locations for 3-62
validity bit for 11-21
external interruption 6-10
clock-comparator 4-45, 6-11
CPU-timer 4-46, 6-11
direct conditions 6-10
emergency-signal 6-11
ETR 6-12
external-call 6-12
external interruption (continued)
  interrupt-key 6-12
  malfunction-alert 6-12
  mask in PSW 4-5
  parameter 6-10
    assigned storage locations for 3-58
  pending conditions 6-10
  priority of conditions 6-11
  service-signal 6-12
external time reference (ETR) 2-6
external-time-reference (ETR) facility 1-13
externally initiated functions 4-47
  I/O 17-17
EXTRACT ACCESS instruction 7-120
EXTRACT AND SET EXTENDED AUTHORITY instruction 10-21
EXTRACT FPC instruction 19-33
EXTRACT PRIMARY ASN AND INSTANCE instruction 10-21
EXTRACT PRIMARY ASN instruction 10-21
EXTRACT PSW instruction 7-120
EXTRACT SECONDARY ASN AND INSTANCE instruction 10-22
EXTRACT SECONDARY ASN instruction 10-22
EXTRACT STACKED REGISTERS instruction 10-22
EXTRACT STACKED STATE instruction 10-24
extraction-authority-control bit 5-25

F
failing-storage address 11-23
  assigned storage locations for 3-62
  in ESW 16-32, 16-38
    as result of channel-control check 16-27
    as result of channel-data check 16-27
  validity bit for 11-21
  validity flag for (in ERW) 16-37
failing-storage-address format
  indicator for (in ERW) 16-37
fetch-only bit 5-55
fetch protection 3-9
  bit in storage key 3-8
  override-control bit 3-11
fetch reference 5-96
  access exceptions for 6-39
fetching
  handling of invalid CBC in storage keys during 11-8
  of ART-table and DAT-table entries 5-94
  of instructions 5-93
  of PSWs during interruptions 5-101
  of storage operands 5-96
FICON I/O interface
  publication referenced xxii
FIEBR (LOAD FP INTEGER) BFP instruction 19-35
FIER (LOAD FP INTEGER) HFP instruction 18-15
FIXBR (LOAD FP INTEGER) BFP instruction 19-35
FIXR (LOAD FP INTEGER) HFP instruction 18-15
floating interruption conditions 6-6, 11-23
  clearing of 4-52
floating point
  See also exponent
    binary (BFP) 9-1
    binary data format 19-4
      conversion
        between formats 19-7
        conversion from binary integer 18-11, 19-26
        conversion to binary integer 18-11, 19-26
      data
        lengthening format of 18-15, 19-37
        shortening format of 18-17, 19-38
      data class 19-5
    hexadecimal (HFP) 9-1
    hexadecimal data format 18-3
    instructions 9-1
    numbers 19-4
    registers 2-3, 9-2
      clearing of 9-11
      save areas for 3-62
      validity bit for 11-21
    shifting
      See normalization
    floating-point-control (FPC) register 19-2
      validity bit for 11-22
    floating-point control register
      save areas for 3-63
format
  address 3-2
  binary-floating-point data 19-4
CCW
  See CCW format control
format (continued)
decimal data 8-1
error
  in DAT-table entry 6-36
  in PSW 6-9
general data 7-2
hexadecimal-floating-point data 18-3
information 3-2
instruction 5-3
PSW 4-5
format-0 and format-1 CCWs 15-27
format-2-IDA W control 15-25
format control 15-24
format-1-sibling-descriptor bit 7-75
forward-section-header address 5-80
forward-section validity bit 5-80
FPC (floating-point-control) register 19-2
fraction 18-1
free-pool manipulation
  programming example A-48
freeze 7-80
fullword
  See word
function control (I/O) 16-12
function-pending time 17-3
  in extended measurement word 16-42
  in measurement block 17-5

G
G (giga) xx
general instructions 7-2
  examples A-7
general register
  save areas 3-63
general registers 2-3
    validity bit for 11-21
 glue module 5-19
GMT (Greenwich Mean Time) obsolete term for UTC 4-42
Greenwich Mean Time (GMT) obsolete term for Coordinated Universal Time 4-42
guard digit 18-4

H
halfword 3-3
halfword-concurrent storage references 5-99
halt function 15-15
  bit in SCSW for 16-12
  completion of 15-16
  initiated by HALT SUBCHANNEL 14-6
  path management for 15-15
  pending 16-15
  signaling for 15-16
halt signal 17-12
  issued as part of halt function 15-16
HALT SUBCHANNEL instruction 14-6
  See also halt function
  effect on SCSW count field 15-18
  function initiated by 15-15
  use of after RESET CHANNEL PATH 14-10
HALVE HFP instructions 18-13
  example A-41
HASCE (home address-space-control element) 3-38
HDR (HALVE) HFP instruction 18-13
  example A-41
header entry 5-79
HER (HALVE) HFP instruction 18-13
hex
  See hexadecimal
  hexadecimal (hex) representation 5-6
hexadecimal floating point
    conversion
      examples with instructions A-42
      instructions
        examples A-39
hexadecimal floating-point number
    examples A-5
HFP (hexadecimal floating point) 9-1
    conversion
      basic example A-6
hexadecimal-floating-point number
    examples A-5
HFP data 18-3
    conversion of 9-8
HFP exponent
    overflow
      exception 6-23
    underflow
      exception 6-23
HFP significance
    exception 6-23
HFP square root
    exception 6-23
high-speed data transfer (I/O) 13-3
home address space 3-16, 5-42
  facilities 5-42
home designation-type-control bits 3-38
home private-space-control bit 3-38
home real-space-control bit 3-38
home real-space token origin (HRSTKO) 3-38
home region table
    designation (HRTD) 3-38
    length (HRTL) 3-38
    origin (HRTO) 3-38
home segment table
    designation (HSTD) 3-38
    length (HSTL) 3-38
    origin (HSTO) 3-38
home-space mode 3-35
home space-switch-event-control bit 3-38
home storage-alteration-event-control bit 3-38
home virtual address 3-5
effective address-space-control element for 3-42
HRSD (home real-space designation) 3-38
HRTD (home region-table designation) 3-38
HRTL (home region-table length) 3-38
HRTO (home region-table origin) 3-38
HSCH (HALT SUBCHANNEL) instruction 14-6
HSTD (home segment-table designation) 3-38
HSTL (home segment-table length) 3-38
HSTO (home segment-table origin) 3-38
I field of instruction 5-7
I instruction format 5-5
I/O (input/output) 2-6
basic functions of 15-1
blocking of data for 15-22
effect on CPU timer 4-46
sense data
See sense data
support functions of 17-1
I/O addressing 13-5
I/O commands
See also commands
publication referenced xxi
I/O device
See device
I/O-error alert (in subchannel logout) 16-35
I/O instructions 14-1, 14-2
defered condition code for 16-8
operand access by 14-1
role of in I/O operations 13-6
I/O interface
ESCON publication referenced xxi
OEMI publication referenced xxi
I/O interruption 6-13, 16-1
See also interruption
action for 16-5
masking of 13-9
priority of 16-4
program-controlled interruption
See PCI
I/O-interruption code 6-13
I/O-interruption condition 13-9, 16-2
alert 16-4
intermediate 16-4
primary 13-8, 16-4
secondary 13-8, 16-4
solicited 16-3
unsolicited 16-3
I/O-interruption-identification word
assigned storage locations for 3-61
I/O-interruption parameter
assigned storage locations for 3-61
in ORB 15-22
in PMCW 15-2
used for IPL 17-18
I/O-interruption request
clearing of 13-9
from subchannels 16-5
I/O-interruption subclass 13-9
I/O-interruption subclass mask 6-13, 16-5
relation to priority 16-4
I/O mask in PSW 4-5
I/O operations 13-6
conclusion of
See conclusion of I/O operations
immediate 15-43
initiated indication for 16-11
termination of
See conclusion of I/O operations
I/O-system reset 17-13
as part of subsystem reset 4-52
I/O-interruption code 14-19
interruption-identification word in 14-19
stored by TPI 14-19
I/O-interruption parameter
in I/O-interruption code 14-19
I/O-interruption-identification word 14-19
IAC (INSERT ADDRESS SPACE CONTROL) instruction 10-27
IC (INSERT CHARACTER) instruction 7-121
IC (instruction counter)
See instruction address
ICM (INSERT CHARACTERS UNDER MASK) instruction 7-121
examples A-21
ICMH (INSERT CHARACTERS UNDER MASK) instruction 7-121
ICMY (INSERT CHARACTERS UNDER MASK) instruction 7-121
ICY (INSERT CHARACTER) instruction 7-121
ID
See CPU identification, sense ID
IDA (indirect-data address) 15-36
flag in CCW 15-28
IDAW (indirect-data-address word) 15-36
check (in subchannel logout) 16-33
invalid address of 16-24
invalid address specification in 16-24
invalid address specification of 16-25
idle state for subchannel 16-13
IDTE (INVALIDATE DAT TABLE ENTRY) instruction 10-30
IEEE-exception-condition data exception 6-21, 19-10
IEEE exception conditions 19-15
summary figure 19-15
IEEE standard 1-12
IFCC (interface-control check) 16-28
IIHH (INSERT IMMEDIATE) instruction 7-122
IIHL (INSERT IMMEDIATE) instruction 7-122
IILL (INSERT IMMEDIATE) instruction 7-122
ILC (instruction-length code) 6-7
assigned storage locations for 3-58
for program interruptions 6-14
for supervisor-call interruption 6-47
IML (initial machine loading) controls 12-3
immediate operand 5-7
immediate operation
SLI flag in CCW for 15-31
immediate operation (I/O) 15-43
implicit address translation 3-41
incorrect length (subchannel status) 16-23
for immediate operations 15-31
incorrect-length-indication mode 15-25
incorrect-length-indication-suppression facility 17-24
effect on immediate operation 15-31
incorrect-length-suppression mode 15-25
incorrect state (signal-processor status) 4-62
index
for address generation 5-8
instructions for branching on 7-27
into access list 5-52
into ASN first and second tables 3-24
into authority table 5-27
into entry and linkage tables 5-31
register for 2-3
indicator
check-stop 12-3
load 12-3
manual 12-4
mode 12-2
test 12-5
wait 12-6
indirect-data address
See IDA
indirect-data-address word
See IDAW
indirect storage error 11-20
infinities 19-6
information format 3-2
initial-command-response measurement (I/O) 17-10
initial-command-response time (in measurement block) 17-6
initial CPU reset 4-52
signal-processor order 4-57
initial-machine-loading (IML) controls 12-3
initial program loading
See IPL
initial-status-interruption control 15-24, 16-11
relation to Z bit 16-11
used for IPL 17-18
inoperative (signal-processor status) 4-62
input/output
See I/O
INSERT ADDRESS SPACE CONTROL
instruction 10-27
INSERT CHARACTER instructions 7-121
INSERT CHARACTERS UNDER MASK instruction
examples A-21
INSERT CHARACTERS UNDER MASK
instructions 7-121
INSERT IMMEDIATE instructions 7-122
INSERT PROGRAM MASK instruction 7-122
INSERT PSW KEY instruction 10-28
INSERT STORAGE KEY EXTENDED
instruction 10-28
INSERT VIRTUAL STORAGE KEY instruction 10-29
installation 2-1
instruction address
as a type of address 3-5
handling by DAT 3-35
in entry-table entry 5-35
in PSW 4-7
validity bit for 11-21
instruction-length code
See ILC
instruction-processing damage 11-16
resulting in processing backup 11-18
resulting in processing damage 11-19
instructions
See also instruction lists and page numbers in Appendix B
backing up of 11-18
classes of 2-2
control 10-1
damage to 11-16, 11-19
decimal 8-1
examples A-34
divisible execution of 5-89
ending of 5-20
examples of use A-6
execution of 5-11
fetching of 5-93
access exception for 6-39
PER event for 4-34
PER-event mask for 4-28
floating-point 9-1
format of 5-3
general 7-2
examples A-7
hexadecimal-floating-point
examples A-39
I/O
See I/O instructions
interruptible
See interruptible instructions
length of 5-6
instructions (continued)

list of B-1
modification by EXECUTE instruction 7-119
prefetching of 5-93
privileged 4-6
for control 10-1
semiprivileged 4-6, 10-1
sequence of execution of 5-2
stepping of (rate control) 12-4
effect on CPU state 4-2
effect on CPU timer 4-46
unprivileged 4-6, 7-2

integer
binary 7-3
address as 5-9
conversion from floating point 18-11, 19-26
conversion to floating point 18-11, 19-26
examples A-2
decimal 8-2
integer quotient 19-29
integral boundary 3-3

interface
ESCON I/O
publication referenced xxi
parallel-I/O
OEMI publication referenced xxi
serial-I/O
publication referenced xxi
interface-control check (subchannel status) 16-28
interlocked-update storage reference 5-97
interlocks for virtual storage references 5-90
intermediate interruption condition (I/O) 16-4
intermediate-status bit (I/O) 16-17
interruption conditions 6-1
clearing of 4-51
floating 6-6, 11-23
I/O
See I/O-interruption condition

time related to CPU state 4-2

priority of
See priority
program
See program interruption
program-controlled (I/O)
See PCI
restart 6-47
string
See string of interruptions
supervisor-call 6-47

interruption code 6-5
external 6-10
I/O
See I/O-interruption code
machine-check (MCIC) 3-62, 11-15
program 6-14
summary of 6-2
supervisor-call 6-47

interruption conditions 6-1
clearing of 4-51
floating 6-6, 11-23
I/O
See I/O-interruption condition

interruption parameter
external (assigned storage locations) 3-58
I/O
See I/O-interruption parameter
interruption-response block
See IRB

interruption subclass
See I/O-interruption subclass
invalid
access-list entry 5-55
address 6-15
bit in ASN-first-table entry 3-25
bit in ASN-second-table entry 3-26
bit in linkage-first-table entry 5-34
bit in linkage-second-table entry 5-34
bit in linkage-table entry 5-33
bit in page-table entry 3-41
bit in region-table entry 3-39
bit in segment-table entry 3-40
CBC 11-2
in registers 11-10
invalid (continued)
CBC (continued)
in storage 11-6
in storage keys 11-7
operation code 6-26
order (signal-processor status) 4-62
parameter (signal-processor status) 4-62
translation address 3-49
translation format
exception recognition 3-49
invalid address specification
in channel-program address 16-24
in IDAW 16-25
of data in CCW 16-25
of IDAW 16-24
of TIC CCW 16-24
invalid CCW field
command code 16-24
count 16-24
data address 16-25
suspend flag 16-25
invalid format
of CCW 16-25
of ORB 16-26
invalid sequence of CCWs 16-26
INVALIDATE DAT TABLE ENTRY instruction 10-30
INVALIDATE PAGE TABLE ENTRY instruction 10-34
effect of when CPU is stopped 4-2
inverse move
See MOVE INVERSE instruction, move-inverse facility
IPK (INSERT PSW KEY) instruction 10-28
IPL (initial program loading) 4-54, 17-17
effect on CPU state 4-2
IPM (INSERT PROGRAM MASK) instruction 7-122
IPTE (INVALIDATE PAGE TABLE ENTRY) instruction 10-34
IRB (interruption-response block) 16-6
See also ECW, ERW, ESW, SCSW
storage requirements for 16-11
ISC (I/O-interruption-subclass code) 15-2
ISC (I/O-interruption-subclass code)
  enhanced 14-19
ISKE (INSERT STORAGE KEY EXTENDED) instruction 10-28
isolated state 16-37
IVSK (INSERT VIRTUAL STORAGE KEY) instruction 10-29

K
K (kilo) xx
KDB (COMPARE AND SIGNAL) BFP instruction 19-24
KDBR (COMPARE AND SIGNAL) BFP instruction 19-24
KEB (COMPARE AND SIGNAL) BFP instruction 19-24
KEBR (COMPARE AND SIGNAL) BFP instruction 19-24
key
  access
    See access key
  manual
    See manual operation
  PSW
    See PSW key
  storage
    See storage key
  subchannel
    See subchannel key
key check (in subchannel logout) 16-32
key-controlled protection 3-9
  exception for 6-28
key mask
  authorization 5-35
  entry 5-35
  PSW (PKM) 5-26
KIMD (COMPUTE INTERMEDIATE MESSAGE DIGEST) instruction 7-84
KLMD (COMPUTE LAST MESSAGE DIGEST) instruction 7-84
KM (CIPHER MESSAGE) instruction 7-35
KMAC (COMPUTE MESSAGE AUTHENTICATION CODE) instruction 7-91
KMC (CIPHER MESSAGE WITH CHAINING) instruction 7-35
KXBR (COMPARE AND SIGNAL) BFP instruction 19-24

L
L (LOAD) binary instruction 7-123
  example A-22
L fields of instruction 5-7
LA (LOAD ADDRESS) instruction 7-124
  examples A-22
LAE (LOAD ADDRESS EXTENDED) instruction 7-124
LAM (LOAD ACCESS MULTIPLE) instruction 7-123
LAMY (LOAD ACCESS MULTIPLE) instruction 7-123
LARL (LOAD ADDRESS RELATIVE LONG) instruction 7-125
LASP (LOAD ADDRESS SPACE PARAMETERS) instruction 10-36
last-path-used mask
  See LPUM
late exception recognition 6-9
LAY (LOAD ADDRESS) instruction 7-124
LB (LOAD BYTE) instruction 7-126
LCDBR (LOAD COMPLEMENT) BFP instruction 19-34
LCDR (LOAD COMPLEMENT) HFP instruction 18-14
LCEBR (LOAD COMPLEMENT) BFP instruction 19-34
LCER (LOAD COMPLEMENT) HFP instruction 18-14
LCGFR (LOAD COMPLEMENT) binary instruction 7-126
LCGR (LOAD COMPLEMENT) binary instruction 7-126
LCR (LOAD COMPLEMENT) binary instruction 7-126
LCTL (LOAD CONTROL) instruction 10-45
LCTLG (LOAD CONTROL) instruction 10-45
LCXBR (LOAD COMPLEMENT) BFP instruction 19-34
LCXR (LOAD COMPLEMENT) HFP instruction 18-14
LD (LOAD) floating-point instruction 9-10
LDE (LOAD LENGTHENED) HFP instruction 18-15
LDEB (LOAD LENGTHENED) BFP instruction 19-37
LDEBR (LOAD LENGTHENED) BFP instruction 19-37
LDER (LOAD LENGTHENED) HFP instruction 18-15
LDR (LOAD) floating-point instruction 9-10
LDXBR (LOAD ROUNDED) BFP instruction 19-38
LDXR (LOAD ROUNDED) HFP instruction 18-17
LDY (LOAD) floating point instruction 9-11
LE (LOAD) floating-point instruction 9-10
LEDBR (LOAD ROUNDED) BFP instruction 19-38
LEDR (LOAD ROUNDED) HFP instruction 18-17
length
field 3-2
instruction 5-6
of BFP data
  decreasing 19-38
  increasing 19-37
of HFP data
  decreasing 18-17
  increasing 18-15
register-operand 5-7
second operand same as first 5-7
variable (storage operand) 5-7
LER (LOAD) floating-point instruction 9-10
LEXBR (LOAD ROUNDED) BFP instruction 19-38
LEXR (LOAD ROUNDED) HFP instruction 18-17
LDY (LOAD) floating point instruction 9-11
LFPC (LOAD FPC) instruction 19-36
LFT (linkage first table) 5-33
LFTD (linkage-first-table designation) 5-33
LFTL (linkage-first-table length) 5-33
LFTO (linkage-first-table origin) 5-33
LFX (linkage first index)
  invalid bit 5-34
  translation exception 6-23
LG (LOAD) binary instruction 7-123
LGB (LOAD BYTE) instruction 7-126
LGF (LOAD) binary instruction 7-123
LGFR (LOAD) binary instruction 7-123
LGH (LOAD HALFWORD) instruction 7-127
LGHI (LOAD HALFWORD IMMEDIATE) instruction 7-127
LGR (LOAD) binary instruction 7-123
LH (LOAD HALFWORD) instruction 7-127
examples A-23
LHI (LOAD HALFWORD IMMEDIATE) instruction 7-127
LHY (LOAD HALFWORD) instruction 7-127
LIFO (last in first out) queueing example for lock and unlock A-46
light
See indicator
limit mode (I/O) 15-2
link information
  for BRANCH AND LINK instruction 7-22
  for BRANCH AND SAVE AND SET MODE instruction 7-23
  for BRANCH AND SAVE instruction 7-23
linkage first table (LFT) 5-33
  designation (LFTD) 5-33
  length (LFTL) 5-33
  origin (LFTO) 5-33
linkage for subroutines 5-11
linkage index (LX) 5-31
linkage second table (LST) 5-34
  origin (LSTO) 5-34
linkage stack 5-69, 5-78
  associated PER events 5-74
  associated trace entries 5-74
  branch state entry 10-10
  entry address 5-77
  entry descriptor 5-78
  entry-type code 5-78
  handling of information in header entry 5-79
  instructions 5-70
  introduction 5-75
  next-entry size 5-79
  operations 5-75
  control 5-77
  program-call state entry 10-67
  remaining free space 5-78
  section 5-75
  identification 5-78
  state entry 5-80
  trailer entry 5-80
linkage-stack functions 5-70
linkage table (LT) 5-33
  designation (LTD) 5-32
  length (LTL) 5-33
  origin (LTO) 5-33
list-directed IPL (list-directed initial program loading) 17-20
little endian 7-132
LLGC (LOAD LOGICAL CHARACTER) instruction 7-128
LLGF (LOAD LOGICAL) instruction 7-128
LLGFR (LOAD LOGICAL) instruction 7-128
logical string assist 1-10
logically partitioned (LPAR) mode 1-14, 1-16
logout
  fixed
    assigned storage locations for 3-63
    machine-check 11-25
  subchannel (I/O) 16-32
long binary-floating-point number 19-4
long hexadecimal-floating-point number 18-3
long I/O block 16-23
loop control 5-11
loop of interruptions
  See string of interruptions
low-address protection 3-12
  control bit 3-12
  exception for 6-28
LPAR (logically partitioned) mode 1-14, 1-16
LPDBR (LOAD POSITIVE) BFP instruction 19-38
LPDR (LOAD POSITIVE) HFP instruction 18-16
LPEBR (LOAD POSITIVE) BFP instruction 19-38
LPER (LOAD POSITIVE) HFP instruction 18-16
LPGFR (LOAD POSITIVE) binary instruction 7-131
LPGR (LOAD POSITIVE) binary instruction 7-131
LPM (logical-path mask) 15-4, 15-25
  effect on system performance of 15-11
  used for IPL 17-18
LPQ (LOAD PAIR FROM QUADWORD)
  instruction 7-131
LPRT (LOAD POSITIVE) binary instruction 7-131
LPSW (LOAD PSW) instruction 10-45
LPSWE (LOAD PSW EXTENDED) instruction 10-46
LPUM (last-path-used mask) 15-4
  field-validity flag for (in subchannel logout) 16-34
  in ESW 16-34
LPXBR (LOAD POSITIVE) BFP instruction 19-38
LPXR (LOAD POSITIVE) HFP instruction 18-16
LR (LOAD) binary instruction 7-123
LRAG (LOAD REAL ADDRESS) instruction 10-52
LX (linkage index) 5-31
  invalid bit 5-33
  translation exception 6-24
LY (LOAD) binary instruction 7-123
LY (LOAD) binary instruction 7-123
LTD (LOAD LENGTHENED) HFP instruction 18-15
LXBR (LOAD) floating-point instruction 9-10
LXDR (LOAD LENGTHENED) HFP instruction 18-15
LXDR (LOAD LENGTHENED) HFP instruction 18-15
LZER (LOAD ZERO) floating-point instruction 9-11
LZXR (LOAD ZERO) floating-point instruction 9-11
MZ (mega) xx
M (MULTIPLY) binary instruction 7-148
  example A-27
machine check 11-1
  See also malfunction
  handling of malfunction detected as part of I/O 11-5
  interruption 6-13, 11-11
  action 11-12
  code (MCIC) 3-62, 11-15
  floating conditions 11-24
  machine check interruption 11-24
  mask in PSW 4-6
  subclass masks in control register 11-24
  logout 11-25
  mask
    in PSW 4-6
  machine-check architectural-mode identification 3-60
  machine-type number (in CPU ID) 10-108
MAD (MULTIPLY AND ADD) HFP instruction 18-19
MADB (MULTIPLY AND ADD) BFP instruction 19-41
MADB (MULTIPLY AND ADD) BFP instruction 19-41
MADR (MULTIPLY AND ADD) HFP instruction 18-19
MAE (MULTIPLY AND ADD) HFP instruction 18-19
MAEB (MULTIPLY AND ADD) BFP instruction 19-41
MAEBR (MULTIPLY AND ADD) BFP instruction 19-41
MAER (MULTIPLY AND ADD) HFP instruction 18-19
main storage 3-2
See also storage
effect of power-on reset on 4-53
shared (in multiprocessing) 4-55
malfunction 11-1
at channel subsystem 16-27
at I/O device 16-28
correction of 11-2
effect on manual operation 12-1
from DIAGNOSE instruction 10-20
indication of 11-5
machine-check handling for when detected as part of I/O 11-5
malfunction alert (external interruption) 6-12
when entering check-stop state 11-11
manual indicator 12-4
See also stopped state
manual operation 12-1
controls
address-compare 12-1
alter-and-display 12-2
IML 12-3
load-unit-address 12-4
power 12-4
rate 12-4
TOD-clock 12-6
effect on CPU signaling 4-60
keys
interrupt 12-3
load-clear 12-3
load-clear-list-directed 12-3
load-normal 12-3
load-with-dump 12-3
restart 12-4
start 12-5
stop 12-5
store-status 12-5
system-reset-clear 12-5
system-reset-normal 12-5
masks 6-6
See also I/O interruption, interruption
for BFP arithmetic exceptions 19-3
for IEEE exception conditions 19-3
in BRANCH ON CONDITION instruction 7-25
in BRANCH RELATIVE ON CONDITION instruction 7-29
in COMPARE LOGICAL CHARACTERS UNDER MASK instruction 7-57
in INSERT CHARACTERS UNDER MASK instruction 7-121
in PSW 4-5
masks (continued)
in STORE CHARACTERS UNDER MASK instruction 7-182
in TEST UNDER MASK instruction 7-192
monitor 6-25
path-management 15-2, 15-25
PER-event 4-28
program-interruption 6-14
subclass
See subclass-mask bits
maximum negative number 7-3
MBA (measurement-block address) 15-8
MC (MONITOR CALL) instruction 7-133
MCIC (machine-check-interruption code) 3-62, 11-15
MD (MULTIPLY) HFP instruction 18-18
MDB (MULTIPLY) BFP instruction 19-39
MDBR (MULTIPLY) BFP instruction 19-39
MDE (MULTIPLY) HFP instruction 18-18
MDEB (MULTIPLY) BFP instruction 19-39
MDEBR (MULTIPLY) BFP instruction 19-39
MDER (MULTIPLY) HFP instruction 18-18
MDR (MULTIPLY) HFP instruction 18-18
example A-41
ME (MULTIPLY) HFP instruction 18-18
measurement
block (I/O)
address 17-8
format 17-7
origin 17-7
device-connect-time 17-10
extended-measurement-word 17-11
measurement-block update (I/O) 17-3
measurement block (I/O) 17-3
data check 16-33
index 15-6
key (MBK)
used as access key 3-9
multiple use of 15-11
program check 16-33
protection check 16-33
update enable 15-3
measurement-block-format control (I/O) 15-7
measurement data (I/O)
accumulated 17-3
effect of CSCH on 14-5
effect of HSCH on 14-6
measurement-mode control (I/O) 15-3
MEE (MULTIPLY) HFP instruction 18-18
MEEB (MULTIPLY) BFP instruction 19-39
MEEBR (MULTIPLY) BFP instruction 19-39
MEER (MULTIPLY) HFP instruction 18-18
MER (MULTIPLY) HFP instruction 18-18
message byte (in EDIT) 8-7
MGHI (MULTIPLY HALFWORD IMMEDIATE) instruction 7-149
MH (MULTIPLY HALFWORD) instruction 7-149
example A-27
MHI (MULTIPLY HALFWORD IMMEDIATE)
instruction 7-149
ML (MULTIPLY LOGICAL) instruction 7-150
MLG (MULTIPLY LOGICAL) instruction 7-150
MLGR (MULTIPLY LOGICAL) instruction 7-150
MLR (MULTIPLY LOGICAL) instruction 7-150
mode
access-register 3-35
addressing
See addressing mode
architectural
See architectural mode
burst (channel-path operation) 13-3
byte-multiplex (channel-path operation) 13-3
home-space 3-35
incorrect-length-indication 15-25
incorrect-length-suppression 15-25
indicator
architectural 12-2
multipath
See multipath mode
primary-space 3-35
real 3-35
requirements for semiprivileged instructions 5-25
rounding 19-7
secondary-space 3-35
single-path 15-3, 15-21
translation 3-35
mode-trace-control bit 4-14
model number (in CPU ID) 10-108
modifiable area (in linkage-stack state entry) 5-83
modification control 15-23
MODIFY STACKED STATE instruction 10-52
MODIFY SUBCHANNEL instruction 14-7
MONITOR CALL instruction 7-133
monitor-class number 6-25
assigned storage locations for 3-59
monitor code 6-25
assigned storage locations for 3-61
monitor event 6-25
monitor masks 6-25
monitoring
See also measurement
channel-subsystem 17-1
for PER events
See PER
with MONITOR CALL 6-25, 7-133
MOVE instructions 7-133
examples A-21, A-23
move-inverse facility 7-134
MOVE INVERSE instruction 7-134
example A-24
MOVE LONG EXTENDED instruction 7-139
MOVE LONG instruction 7-134
examples A-25
MOVE LONG UNICODE instruction 7-142
MOVE NUMERICs instruction 7-146
example A-25
MOVE PAGE instruction 10-54
MOVE STRING instruction 7-146
example A-26
MOVE TO PRIMARY instruction 10-56
MOVE TO SECONDARY instruction 10-56
MOVE WITH DESTINATION KEY instruction 10-58
MOVE WITH KEY instruction 10-59
MOVE WITH OFFSET instruction 7-147
example A-26
MOVE WITH SOURCE KEY instruction 10-60
MOVE ZONES instruction 7-148
example A-27
MP (MULTIPLY DECIMAL) instruction 8-11
example A-36
MR (MULTIPLY) binary instruction 7-148
example A-27
MS (MULTIPLY SINGLE) instruction 7-151
MSCH (MODIFY SUBCHANNEL) instruction 14-7
MSD (MULTIPLY AND SUBTRACT) HFP
instruction 18-20
MSDB (MULTIPLY AND SUBTRACT) BFP
instruction 19-41
MSDBR (MULTIPLY AND SUBTRACT) BFP
instruction 19-41
MSDR (MULTIPLY AND SUBTRACT) HFP
instruction 18-20
MSE (MULTIPLY AND SUBTRACT) HFP
instruction 18-20
MSEB (MULTIPLY AND SUBTRACT) BFP
instruction 19-41
MSEBR (MULTIPLY AND SUBTRACT) BFP
instruction 19-41
MSER (MULTIPLY AND SUBTRACT) HFP
instruction 18-20
MSG (MULTIPLY SINGLE) instruction 7-151
MSGF (MULTIPLY SINGLE) instruction 7-151
MSGFR (MULTIPLY SINGLE) instruction 7-151
MSGR (MULTIPLY SINGLE) instruction 7-151
MSR (MULTIPLY SINGLE) instruction 7-150
MSTA (MODIFY STACKED STATE) instruction 10-52
MSY (MULTIPLY SINGLE) instruction 7-151
multipath mode 15-3
entering 15-21
multiple-access storage references 5-98
MULTIPLY AND ADD BFP instructions 19-41
MULTIPLY AND ADD HFP instructions 18-19
MULTIPLY AND SUBTRACT BFP instructions 19-41
MULTIPLY AND SUBTRACT HFP instructions 18-20
MULTIPLY BFP instructions 19-39
MULTIPLY binary instructions 7-148
examples A-27
MULTIPLY DECIMAL instruction  8-11
  example A-36
MULTIPLY HALFWORD IMMEDIATE instruction  7-149
MULTIPLY HALFWORD instruction  7-149
  example A-27
MULTIPLY HFP instructions  18-18
  example A-41
MULTIPLY LOGICAL instructions  7-150
MULTIPLY SINGLE instructions  7-150
multiprocessing  4-55
  manual operations for  12-6
  programming considerations for  8-3, A-43
  programming examples A-43
  timing-facility interruptions for  4-44
  TOD clock for  4-40
multiprogramming examples A-43
MVC (MOVE) instruction  7-133
  examples A-21, A-23
MVCDFK (MOVE WITH DESTINATION KEY)  instruction 10-58
MVCIN (MOVE INVERSE) instruction  7-134
  example A-24
MVCK (MOVE WITH KEY) instruction  10-59
MVCL (MOVE LONG) instruction  7-134
  examples A-25
MVCLE (MOVE LONG EXTENDED) instruction  7-139
MVCLUD (MOVE LONG UNICODE) instruction  7-142
MVCMP (MOVE TO PRIMARY) instruction  10-56
MVCS (MOVE TO SECONDARY) instruction  10-56
MVCSK (MOVE WITH SOURCE KEY)  instruction 10-60
MVI (MOVE) instruction  7-133
  example A-24
MVIY (MOVE) instruction  7-133
MVN (MOVE NUMERICS) instruction  7-146
  example A-25
MVO (MOVE WITH OFFSET) instruction  7-147
  example A-26
MVPG (MOVE PAGE) instruction  10-54
MVST (MOVE STRING) instruction  7-146
  example A-26
MVZ (MOVE ZONES) instruction  7-148
  example A-27
MXBR (MULTIPLY) BFP instruction  19-39
MXD (MULTIPLY) HFP instruction  18-18
MXDB (MULTIPLY) BFP instruction  19-39
MXDBR (MULTIPLY) BFP instruction  19-39
MXDR (MULTIPLY) HFP instruction  18-18
MXR (MULTIPLY) HFP instruction  18-18
NC (AND) instruction  7-21
near-valid CBC  11-2
  in storage  11-5
negative zero
  decimal  8-2
  example A-5
new PSW  4-3
  assigned storage locations for  3-62
  fetched during interruption  6-2
next-entry size (in linkage stack)  5-79
NG (AND) instruction  7-20
NGR (AND) instruction  7-20
NI (AND) instruction  7-20
  example A-8
NIHH (AND IMMEDIATE) instruction  7-21
NIHL (AND IMMEDIATE) instruction  7-21
NILH (AND IMMEDIATE) instruction  7-21
NILL (AND IMMEDIATE) instruction  7-21
NIY (AND) instruction  7-21
no-operation
  instruction (BRANCH ON CONDITION)  7-26
  instruction (BRANCH RELATIVE ON CONDITION)  7-29
node (of tree structure)  7-205
noninterlocked-update storage reference  5-97
nonnumeric entities
  binary  19-6
  nonvolatile storage  3-2
normalization
  of BFP numbers  19-8
  of HFP numbers  18-3
not-a-number (NaN)  19-6
not operational
  as channel-path state  16-12
  See also path-not-operational bit
  as CPU state  4-60
  as TOD-clock state  4-41
not set (TOD-clock state)  4-40
NR (AND) instruction  7-20
nullification
  exceptions to  5-23
  for exigent machine-check conditions  11-11
  of instruction execution  5-21
  of unit of operation  5-22
numbering
  of addresses (byte locations)  3-2
  of bits  3-2
numbers
  binary  7-3
  examples A-2
  binary-floating-point  19-4
CPU-model  10-108
decimal  8-1
  examples A-4
device  13-5

N
N (AND) instruction  7-20
N condition (I/O)  16-12
NaN (not-a-number)  19-6
numbers (continued)
  hexadecimal 5-6
  hexadecimal-floating-point 18-3
  hexadecimal-floating-point examples A-5
  machine-type 10-108
numeric bits 8-1
  moving of 7-146
Numeric part of PC number 5-82
NY (AND) instruction 7-20

O
  O (OR) instruction 7-151
  OC (OR) instruction 7-152
  OEMI (original equipment manufacturers information) for I/O interface xxi
    publication referenced xxi
  offset of symbol-translation table 7-73
  OG (OR) instruction 7-152
  OGR (OR) instruction 7-151
  OI (OR) instruction 7-152
    example A-28
    example of problem with A-43
  OIHH (OR IMMEDIATE) instruction 7-152
  OIHL (OR IMMEDIATE) instruction 7-152
  OILH (OR IMMEDIATE) instruction 7-153
  OILL (OR IMMEDIATE) instruction 7-153
  OIY (OR) instruction 7-152
  old PSW 6-2
    assigned storage locations for 3-62
  one's complement binary notation 7-3
    used for SUBTRACT LOGICAL instruction 7-189
    used for SUBTRACT LOGICAL WITH BORROW instruction 7-190
  op code
    See operation code
  operand 5-3
    access identification 3-60
    access of 5-96
      for I/O instructions 14-1
    address generation for 5-8
    exception 6-25
    immediate 5-7
    length of 5-3
    overlap of
      for decimal instructions 8-3
      for general instructions 7-2
    register for 5-6
    sequence of references for 5-96
    storage 5-7
    types of (fetch, store, update) 5-96
    used for result 5-3
  operating state 4-1, 4-2
  operation
    I/O
      See I/O operations
operation (continued)
  unit of 5-21
  operation code (op code) 5-2
    invalid 6-26
  operation exception 6-26
  operation-request block
    See ORB
  operator facilities 2-7, 12-1
    basic 12-1
  operator intervening (signal-processor status) 4-62
  OR (OR) instruction 7-151
  OR IMMEDIATE instructions 7-152
  OR instructions 7-151
    example of problem with OR immediate A-43
    examples A-28
  ORB (operation-request block) 15-22
    CSS priority 15-26
  extension control in 15-25
  interruption parameter in 15-22
  invalid 16-26
  logical-path mask (LPM) in 15-25
  orders (I/O) 13-6, 15-29
  orders (signal-processor) 4-56
    conditions precluding response to 4-59
    CPU reset 4-57
    emergency signal 4-56
    external call 4-56
    initial CPU reset 4-57
    restart 4-56
    sense 4-56
    set architecture 4-58
    set prefix 4-57
    start 4-56
    stop 4-56
    stop and store status 4-57
    store status at address 4-57
  overflow
    binary 7-4
      example A-2
    decimal 6-21
    exponent
      See exponent overflow
    fixed-point 6-22, 7-4
    in CRW 17-26
  overlap
    destructive 7-135, 7-139, 7-144
    operand 5-91
      for decimal instructions 8-3
      for general instructions 7-2
    operation 5-89
  OY (OR) instruction 7-152

P
  P (peta) xx
<table>
<thead>
<tr>
<th>Term</th>
<th>Page/Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACK ASCII instruction</td>
<td>7-154</td>
</tr>
<tr>
<td>PACK instruction</td>
<td>7-153</td>
</tr>
<tr>
<td>example</td>
<td>A-28</td>
</tr>
<tr>
<td>PACK UNICODE instruction</td>
<td>7-155</td>
</tr>
<tr>
<td>packed decimal numbers</td>
<td>8-1</td>
</tr>
<tr>
<td>conversion of to zoned format</td>
<td>7-202</td>
</tr>
<tr>
<td>conversion to from zoned format</td>
<td>7-153</td>
</tr>
<tr>
<td>examples</td>
<td>A-4</td>
</tr>
<tr>
<td>padding byte</td>
<td></td>
</tr>
<tr>
<td>for COMPARE LOGICAL LONG EXTENDED instruction</td>
<td>7-61</td>
</tr>
<tr>
<td>for COMPARE LOGICAL LONG instruction</td>
<td>7-58</td>
</tr>
<tr>
<td>for MOVE LONG EXTENDED instruction</td>
<td>7-139</td>
</tr>
<tr>
<td>for MOVE LONG instruction</td>
<td>7-135</td>
</tr>
<tr>
<td>page frame real address (PFRA)</td>
<td>3-41</td>
</tr>
<tr>
<td>PAGE IN instruction</td>
<td>10-61</td>
</tr>
<tr>
<td>page index (PX)</td>
<td>3-34</td>
</tr>
<tr>
<td>page-invalid bit (in page-table entry)</td>
<td>3-41</td>
</tr>
<tr>
<td>PAGE OUT instruction</td>
<td>10-62</td>
</tr>
<tr>
<td>page protection</td>
<td>3-11</td>
</tr>
<tr>
<td>bit for in page-table entry</td>
<td>3-41</td>
</tr>
<tr>
<td>bit for in segment-table entry</td>
<td>3-40</td>
</tr>
<tr>
<td>exception for</td>
<td>6-28</td>
</tr>
<tr>
<td>page frame</td>
<td>3-33</td>
</tr>
<tr>
<td>page table</td>
<td>3-41</td>
</tr>
<tr>
<td>designation</td>
<td>3-40</td>
</tr>
<tr>
<td>lookup</td>
<td>3-49</td>
</tr>
<tr>
<td>origin (PTO)</td>
<td>3-40</td>
</tr>
<tr>
<td>page-translation exception</td>
<td>6-26</td>
</tr>
<tr>
<td>as an access exception</td>
<td>6-36, 6-43</td>
</tr>
<tr>
<td>PALB (PURGE ALB) instruction</td>
<td>10-91</td>
</tr>
<tr>
<td>PAM (path-available mask)</td>
<td>15-7</td>
</tr>
<tr>
<td>effect of reconfiguration on</td>
<td>15-11</td>
</tr>
<tr>
<td>effect of resetting on</td>
<td>15-11</td>
</tr>
<tr>
<td>effect on allegiance of</td>
<td>15-11</td>
</tr>
<tr>
<td>parallel-I/O channel-to-channel adapter publication referenced</td>
<td>xxi</td>
</tr>
<tr>
<td>parallel-I/O interface</td>
<td>13-3</td>
</tr>
<tr>
<td>OEMI publication referenced</td>
<td>xxi</td>
</tr>
<tr>
<td>parameter</td>
<td></td>
</tr>
<tr>
<td>external-interruption</td>
<td>6-10</td>
</tr>
<tr>
<td>assigned storage locations for</td>
<td>3-58</td>
</tr>
<tr>
<td>I/O-interruption</td>
<td></td>
</tr>
<tr>
<td>See I/O-interruption parameter</td>
<td></td>
</tr>
<tr>
<td>register for SIGNAL PROCESSOR</td>
<td>4-57, 10-106</td>
</tr>
<tr>
<td>translation</td>
<td>3-35</td>
</tr>
<tr>
<td>parity bit</td>
<td>11-2</td>
</tr>
<tr>
<td>partial completion of instruction execution</td>
<td>5-21</td>
</tr>
<tr>
<td>PASCE (primary address-space-control element)</td>
<td>3-36</td>
</tr>
<tr>
<td>PASN (primary address-space number)</td>
<td>3-17</td>
</tr>
<tr>
<td>in trace entry</td>
<td>4-26</td>
</tr>
<tr>
<td>PASTE (primary AST entry)</td>
<td>5-32</td>
</tr>
<tr>
<td>PASTEIN (primary AST-entry instance number)</td>
<td>3-19</td>
</tr>
<tr>
<td>PASTEO (primary-AST-entry origin)</td>
<td>5-32, 5-52</td>
</tr>
<tr>
<td>path</td>
<td></td>
</tr>
<tr>
<td>See channel path</td>
<td></td>
</tr>
<tr>
<td>path available for selection</td>
<td>15-13</td>
</tr>
<tr>
<td>path management</td>
<td>13-6</td>
</tr>
<tr>
<td>for clear function</td>
<td>15-14</td>
</tr>
<tr>
<td>for halt function</td>
<td>15-15</td>
</tr>
<tr>
<td>for start function and resume function</td>
<td>15-18</td>
</tr>
<tr>
<td>path-management-control word</td>
<td></td>
</tr>
<tr>
<td>See PMCW</td>
<td></td>
</tr>
<tr>
<td>path-management masks</td>
<td></td>
</tr>
<tr>
<td>last-path-used mask</td>
<td></td>
</tr>
<tr>
<td>See LPUM</td>
<td></td>
</tr>
<tr>
<td>logical-path mask</td>
<td></td>
</tr>
<tr>
<td>See LPM</td>
<td></td>
</tr>
<tr>
<td>path-available mask</td>
<td></td>
</tr>
<tr>
<td>See PAM</td>
<td></td>
</tr>
<tr>
<td>path-installed mask</td>
<td></td>
</tr>
<tr>
<td>See PIM</td>
<td></td>
</tr>
<tr>
<td>path-not-operational mask</td>
<td></td>
</tr>
<tr>
<td>See PNOM</td>
<td></td>
</tr>
<tr>
<td>path-operational mask</td>
<td></td>
</tr>
<tr>
<td>See POM</td>
<td></td>
</tr>
<tr>
<td>path-not-operational bit (N) in SCSW</td>
<td>16-12</td>
</tr>
<tr>
<td>path-not-operational condition</td>
<td>15-4</td>
</tr>
<tr>
<td>path verification required</td>
<td></td>
</tr>
<tr>
<td>indicator for (in ERW)</td>
<td>16-37</td>
</tr>
<tr>
<td>pattern (in EDIT)</td>
<td>8-7</td>
</tr>
<tr>
<td>PC (PROGRAM CALL) instruction</td>
<td>10-63</td>
</tr>
<tr>
<td>PC-cp (PROGRAM CALL instruction, to current primary)</td>
<td>10-67</td>
</tr>
<tr>
<td>PC number</td>
<td>10-64</td>
</tr>
<tr>
<td>in linkage-stack state entry</td>
<td>5-82</td>
</tr>
<tr>
<td>in trace entry</td>
<td>4-25</td>
</tr>
<tr>
<td>translation</td>
<td>5-31</td>
</tr>
<tr>
<td>PC-ss (PROGRAM CALL instruction, with space switching)</td>
<td>10-68</td>
</tr>
<tr>
<td>PC-translation-specification exception</td>
<td>6-26</td>
</tr>
<tr>
<td>PC-type bit</td>
<td>5-35</td>
</tr>
<tr>
<td>PCI (program-controlled interruption)</td>
<td>15-35</td>
</tr>
<tr>
<td>as flag in CCW</td>
<td>15-28</td>
</tr>
<tr>
<td>intermediate interruption condition for</td>
<td>16-17</td>
</tr>
<tr>
<td>subchannel status for</td>
<td>16-23</td>
</tr>
<tr>
<td>pending channel reports (effect of I/O-system reset on)</td>
<td>17-16</td>
</tr>
<tr>
<td>pending interruption</td>
<td></td>
</tr>
<tr>
<td>See interruption pending</td>
<td></td>
</tr>
<tr>
<td>PER (program-event recording)</td>
<td>4-27</td>
</tr>
<tr>
<td>access identification</td>
<td>3-59, 4-31</td>
</tr>
<tr>
<td>address</td>
<td>4-31</td>
</tr>
<tr>
<td>assigned storage locations for</td>
<td>3-59</td>
</tr>
<tr>
<td>address-space-control element (ASCE) identification</td>
<td>4-30</td>
</tr>
<tr>
<td>ASCE (address-space-control element) identification</td>
<td>4-30</td>
</tr>
<tr>
<td>ATMID (addressing-and-translation-mode identification)</td>
<td>4-30</td>
</tr>
</tbody>
</table>
PER (program-event recording) (continued)
code 4-30
  assigned storage locations for 3-59
  events 4-27
  extensions 1-13
  instruction-fetching event 4-34
masks
  bit in PSW 4-5
  PER-event 4-28
  priority of indication 4-32
  program-interruption condition 6-27
  storage-alteration event 4-35
  storage-area designation 4-33
  ending address 4-28
  starting address 4-28
  wraparound 4-33
  store-using-real-address event 4-35
  successful-branching event 4-34
PERFORM LOCKED OPERATION instruction 7-156
  example A-50
PFRA (page-frame real address) 3-41
PGIN (PAGE IN) instruction 10-61
PGOUT (PAGE OUT) instruction 10-62
piecemeal steps of instruction execution 5-89
PIM (path-installed mask) 15-6
PKA (PACK ASCII) instruction 7-154
PKM (PSW-key mask) 5-26
PKU (PACK UNICODE) instruction 7-155
PLO (PERFORM LOCKED OPERATION) instruction 7-156
  example A-50
PMCW (path-management-control word) 15-2
  channel-path identifiers (CHPID) in 15-7
PNOM (path-not-operational mask) 15-4
  effect on POM of 15-11
  indicated in SCSW 16-12
  point of damage 11-14
  point of interruption 5-21
  for machine check 11-14
POM (path-operational mask) 15-6
  effect on PNOM of 15-11
POST (SVC)
  example of routine to bypass A-45
postnormalization 18-3
power controls 12-4
power-on reset 4-53
powers of 2
  table of G-1
PR (PROGRAM RETURN) instruction 10-76
PR-cp (PROGRAM RETURN instruction, to current primary) 10-77
PR-ss (PROGRAM RETURN instruction, with space switching) 10-77
PR/SM (Processor Resource/Systems Manager) 1-14, 1-16
precision (floating-point) 9-1
preferred sign codes 8-2
prefetch control 15-24
prefetching
  See also CCW prefetch control
  access exceptions not recognized for 6-39
  channel-control check during 16-27
  channel-data check during 16-27
  handling of invalid CBC in storage keys during 11-8
  of ART-table and DAT-table entries 5-94
  of data for I/O 15-30
  of instructions 5-93
  of operands 5-96
prefix 3-15
  set by signal-processor order 4-57
  store-status save area for 3-63
prefix area 3-15
prenormalization 18-3
primary address space 3-16
primary ASN (PASN) 3-17
  in linkage-stack state entry 5-81
primary-AST entry (PASTE) 5-32
  origin (PASTEo) 5-32, 5-52
primary ASTEIN (PASTEIN)
  in linkage-stack state entry 5-83
primary authority 3-31
  exception 6-27
  primary designation-type-control bits 3-37
  primary interruption condition (I/O) 16-4
  primary-list bit 5-52
  primary private-space-control bit 3-36
  primary real-space-control bit 3-37
  primary real-space token origin (PRSTKO) 3-37
primary region table
  designation (PRTD) 3-36
  length (PRTL) 3-37
  origin (PRTO) 3-36
primary segment table
  designation (PSTD) 3-36
  length (PSTL) 3-37
  origin (PSTO) 3-36
primary-space access-list designation (PSALD) 5-54
primary-space mode 3-35
primary-space-switch-event-control bit 3-36
primary-status bit (I/O) 16-18
primary storage-alteration-event-control bit 3-36
primary subspace-group-control bit 3-36
primary virtual address 3-4
  effective address-space-control element for 3-42
priority
  of access exceptions 6-43
  of ASN-translation exceptions 6-46
  of data exceptions 6-14
  of external-interruption conditions 6-11
  of I/O interruptions 16-4
  of interruptions (CPU) 6-48
priority (continued)
of PER events 4-32
of program-interruption conditions 6-39
   for arithmetic exceptions 6-14
   of subspace-replacement exceptions 6-47
   of trace exceptions 6-47
private bit 5-55
private-space control
effect on
   fetch-protection override 3-11
   low-address protection 3-12
   use of common segments 3-40
private-space-control bit 3-36
   home 3-38
   primary 3-36
   secondary 3-37
privileged instructions 4-6
   control 10-1
   I/O 14-1
privileged-operation exception 6-27
problem state 4-6
   bit in entry-table entry 5-35
   bit in PSW 4-6
   compatibility 1-17
processing backup (synchronous machine-check condition) 11-18
processing damage (synchronous machine-check condition) 11-19
processor
   See CPU
processor-availability facility 1-14
Processor Resource/Systems Manager (PR/SM) 1-14, 1-16
program 5-46
   channel
      See channel program
   exceptions 6-14
   execution of 5-2
   fields of SCHIB modifiable by 15-9
   initial loading by CCW-type IPL 17-18
   initial loading by list-directed IPL 17-20
   initial loading of 4-54, 17-17
   interruption 6-14
      priority of 6-14, 6-39
   mask (in PSW) 4-6
PROGRAM CALL instruction 10-63
   trace entry for 4-25
   type of 5-35
program-call state entry 5-80, 10-67
program check
   as subchannel status 16-24
   measurement-block 16-33
program-controlled interruption (I/O)
   See PCI
program-event recording
   See PER
program events
   See PER events
program mask
   validity bit for 11-21
PROGRAM RETURN instruction 10-76
program-status word
   See PSW
PROGRAM TRANSFER instruction 10-81
   trace entry for 4-25
PROGRAM TRANSFER WITH INSTANCE instruction 10-81
programmable field of TOD clock 4-42
protection (storage) 3-9
   access-list-controlled
      See access-list-controlled protection
during tracing 4-27
   fetch
      See fetch protection
   key-controlled
      See key-controlled protection
   low-address
      See low-address protection
   page
      See page protection
protection check
   as subchannel status 16-26
   measurement-block 16-33
   protection exception 6-28
   as an access exception 6-36, 6-43
   PRSD (primary real-space designation) 3-36
   PRSTKO (primary real-space token origin) 3-37
   PRTD (primary region-table designation) 3-36
   PRTL (primary region-table length) 3-37
   PRTO (primary region-table origin) 3-36
   PSALD (primary-space access-list designation) 5-54
   pseudo AST entry 3-18
   PSTD (primary segment-table designation) 3-36
   PSTL (primary segment-table length) 3-37
   PSTO (primary segment-table origin) 3-36
   PSW (program-status word) 2-3, 4-3
      current 4-3, 5-11
         stored during interruption 6-2
      exceptions associated with 6-9
      format error 6-9
      in linkage-stack state entry 5-81
      in program execution 5-11
      saved 4-51
      store-status save area for 3-63
      validity bits for 11-21
PSW key 4-5
   control bit 5-74
   in entry-table entry 5-75
   in trace entry 4-25
   used as access key 3-9
   validity bit for 11-21
PSW-key mask (PKM) 5-26
control bit 5-74
in linkage-stack state entry 5-81
PT (PROGRAM TRANSFER) instruction 10-81
PT-ep (PROGRAM TRANSFER instruction, to current
primary) 10-82
PT-ss or PT-ss (PROGRAM TRANSFER (WITH
INSTANCE) instruction, with space switching) 10-83
PTI (PROGRAM TRANSFER WITH INSTANCE)
instruction 10-81
PTLB (PURGE TLB) instruction 10-91
PTO (page-table origin) 3-40
publications
other related documents xxi
PURGE ALB instruction 10-91
PURGE TLB instruction 10-91
PX (page index) 3-34
Q
QNaN (quiet NaN) 19-6
quadword-concurrent storage references 5-99
queueing
FIFO
example for lock and unlock A-47
LIFO
example for lock and unlock A-46
quiet NaN (QNaN) 19-6
R
R field of instruction 5-6
radix
binary 9-1
hexadecimal 9-1
rate control 12-4
RCHP (RESET CHANNEL PATH) instruction 14-9
real address 3-4
real mode 3-35
real space
token origin (RSTKO) 3-36
real-space-control bit 3-38
home 3-38
primary 3-37
secondary 3-37
real-space designation (RSD)
home 3-38
primary 3-36
secondary 3-37
real storage 3-4
receiver check (signal-processor status) 4-62
reconfiguration of I/O system 17-22
recovery
as class of machine-check condition 11-12
channel-subsystem 17-24
system 11-16
subclass-mask bit for 11-25
reduced-authority state 10-7
redundancy 11-2
reference
bit in storage key 3-8
multiple-access 5-98
recording 3-14
sequence for storage 5-88
See also sequence
single-access 5-98
region 3-34
region first index (RFX) 3-34
region-first-translation exception 6-28
as an access exception 6-43
region index (RX)
in virtual address 3-34
region-invalid bit (in region-table entry) 3-39
region second index (RSX) 3-34
region-second-translation exception 6-29
as an access exception 6-43
region table
origin (RTO) 3-36
region-table designation (RTD) 3-36
home 3-38
primary 3-36
secondary 3-37
region-table entry (RTE) 3-39
region third index (RTX) 3-34
region-third-translation exception 6-29
as an access exception 6-43
region-translation exception 6-28
register
access 2-4
base-address 2-3
control 2-4
designation of 5-6
floating-point 2-3, 9-2
floating-point-control 19-2
general 2-3
index 2-3
prefix 3-15
save areas for 3-63, 11-22
validation of 11-10
validity bits for 11-21
relative branching 5-10
remainder 19-9
result of DIVIDE TO INTEGER 19-29
remaining free space (in linkage stack) 5-78
remote operating stations 12-1
reporting-source code (RSC) 17-27
reporting-source ID (RSID) 17-28
repressible machine-check conditions 11-12
request-logging-only 16-36
indicator for (in ERW) 16-36
reset 4-47, 17-13
channel-path 17-13
clear 4-53
reset (continued)
   CPU  4-51
   effect on CPU state  4-2
   effect on TOD clock  4-40
   I/O-system  17-13
      as part of subsystem reset  4-52
   initial CPU  4-52
   power on  4-53
   subsystem  4-52
   summary of functions  4-49
   summary of functions performed by manual initiation
      of  4-48
   system-reset-clear key  12-5
   system-reset-normal key  12-5
RESET CHANNEL PATH instruction  14-9
   See also  channel-path-reset function
   function initiated by  15-45
RESET REFERENCE BIT EXTENDED
   instruction  10-91
reset signal (I/O)  17-13
   in channel-path reset  17-13
   in I/O-system reset  17-14, 17-15
   issued as part of RCHP  15-45
resetting event
   See path verification required
resolution
   of clock comparator  4-45
   of CPU timer  4-46
   of TOD clock  4-40
restart
   interruption  6-47
   key  12-4
   signal-processor order  4-56
result operand  5-3
resume function  13-8, 15-18
   See also  start function
   initiated by RESUME SUBCHANNEL  14-10
   path management for  15-19
   pending  16-13
RESUME PROGRAM instruction  10-92
RESUME SUBCHANNEL instruction  14-10
   See also  resume function
   channel-program requirements for  14-11
   count of in measurement block  17-4
   function initiated by  15-18
retry
   CPU  11-2
   I/O command
      See command retry
reusable-ASN bit  3-19
RFX (region first index)  3-34
RI instruction format  5-5
RIE instruction format  5-5
RIL instruction format  5-6
RLL (ROTATE LEFT SINGLE LOGICAL)
   instruction  7-171
RR (RESET REFERENCE BIT EXTENDED) instruction  10-91
RRBE (RESET REFERENCE BIT EXTENDED) instruction
   10-91
RRF instruction format  5-6
RS instruction format  5-6
RSC (reporting-source code)  17-27
RSCH (RESUME SUBCHANNEL) instruction  14-10
RSI instruction format  5-6
RSID (reporting-source ID)  17-28
RSL instruction format  5-6
RSTKO (real-space token origin)  3-36
RSX (region second index)  3-34
RSY instruction format  5-6
RX (region index)  3-34
RX instruction format  5-6
RXE instruction format  5-6
RXY instruction format  5-6
S
S (SUBTRACT) binary instruction  7-187
S instruction format  5-6
SAC (SET ADDRESS SPACE CONTROL)
   instruction  10-95
SACF (SET ADDRESS SPACE CONTROL FAST)
   instruction  10-95
SAL (SET ADDRESS LIMIT) instruction  14-11
SAM24 (SET ADDRESSING MODE) instruction  7-175
SAM31 (SET ADDRESSING MODE) instruction  7-175
SAM64 (SET ADDRESSING MODE) instruction  7-175
   sample count (in ESW)  17-4
SAR (SET ACCESS) instruction  7-174
SASCE (secondary address-space-control element)  3-37
SASN (secondary address-space number)  3-17
   in trace entry  4-26
SASTEIN (secondary-AST-entry instance number)  3-19
save areas for registers 3-63, 11-22
saved PSW 4-51
SCHIB (subchannel-information block) 15-1
as operand of
  MODIFY SUBCHANNEL 14-7
  STORE SUBCHANNEL 14-17
  Measurement-Block Address (MBA) in 15-8
  model-dependent area in 15-8
  path-management-control word (PMCW) in 15-2
  subchannel-status word (SCSW) in 15-8
  summary of modifiable fields in 15-9
SCHM (SET CHANNEL MONITOR) instruction 14-12
SCK (SET CLOCK) instruction 10-96
SCKC (SET CLOCK COMPARATOR) instruction 10-97
SCKPF (SET CLOCK PROGRAMMABLE FIELD) instruction 10-97
SCP-initiated reset 1-14
SCSW (subchannel-status word) 16-6
  activity-control field in 16-13
  CCW address in 16-18
  count in 16-29
  device-status field in 16-23
  function-control field in 16-12
  in IRB 16-6
  in SCHIB 15-8
  status-control field in 16-16
  subchannel-control field in 16-11
  subchannel-status field in 16-23
SD (SUBTRACT NORMALIZED) HFP instruction 18-22
SDB (SUBTRACT) BFP instruction 19-44
SDR (SUBTRACT NORMALIZED) HFP instruction 18-22
SE (SUBTRACT NORMALIZED) HFP instruction 18-22
SEARCH STRING instruction 7-172
  examples A-29
SEARCH STRING UNICODE instruction 7-173
SEB (SUBTRACT) BFP instruction 19-44
SEBR (SUBTRACT) BFP instruction 19-44
secondary address space 3-16
secondary ASN (SASN) 3-17
  control bit 5-75
  in linkage-stack state entry 5-81
secondary ASTEIN (SASTEIN) in linkage-stack state entry 5-83
secondary authority 3-31
  exception 6-30
secondary-CCW address validity (in ERW) 16-37
secondary designation-type-control bits 3-37
secondary error (in subchannel logout) 16-35
secondary interruption condition (I/O) 16-4
secondary private-space-control bit 3-37
secondary real-space-control bit 3-37
secondary real-space token origin (SRSTKO) 3-37
secondary region table
  designation (SRSD) 3-37
  length (SRTL) 3-37
  origin (SRTO) 3-37
secondary segment table
  designation (SSST) 3-37
  length (SSTL) 3-37
  origin (SSTO) 3-37
secondary-space-control bit 3-36, 5-26
secondary-space mode 3-35
secondary-status bit (I/O) 16-18
secondary storage-alteration-event-control bit 3-37
secondary subspace-group-control bit 3-37
secondary virtual address 3-4
  effective address-space-control element for 3-42
  segment 3-34
  segment index (SX) 3-34
  segment-invalid bit (in segment-table entry) 3-40
  segment table 3-40
    length (STL) 3-36
    origin (STO) 3-36
segment-table designation (STD)
  home 3-38
  obtaining of in access-register translation 5-45
  primary 3-36
  secondary 3-37
segment-translation exception 6-30
  as an access exception 6-36, 6-43
self-describing block of I/O data 15-34
semiprivileged
  instructions 4-6
    descriptions of 10-1
    program authorization 5-25
    summary of 5-29
    programs 4-6, 5-25
sense
  as signal-processor order 4-56
sequence
  conceptual 5-89
  instruction-execution 5-2
  of CCWs that is invalid 16-26
  of storage references 5-88
    ART-table and DAT-table entries 5-94
    for floating-point data 9-2
    instructions 5-93
    operands 5-96
    storage keys 5-95
sequence code (in subchannel logout) 16-35
  field-validity flag for 16-34
SER (SUBTRACT NORMALIZED) HFP instruction 18-22
serial-I/O channel-to-channel adapter
  publication referenced xxi
serial-I/O interface 13-2
publication referenced xxi
serialization 5-102
caued by I/O instructions 14-1
channel-program 5-103
CPU 5-102
in completion of store operations 5-97
service-call-logical-processor (SCLP) facility 1-15
service-processor damage 11-18
service processor inoperative (signal-processor status) 4-62
service-signal external interruption 6-12
subclass-mask bit for 6-13
SET ACCESS instruction 7-96
SET ADDRESS LIMIT instruction 14-11
SET ADDRESS SPACE CONTROL FAST instruction 10-95
SET ADDRESS SPACE CONTROL instruction 10-95
SET ADDRESSING MODE instruction 7-175
SET ADDRESSING MODE instructions 7-175
set architecture
signal-processor order 4-58
SET CHANNEL MONITOR instruction 14-12
effect on measurement modes of 17-1
SET CLOCK COMPARATOR instruction 10-97
SET CLOCK instruction 10-96
SET CLOCK PROGRAMMABLE FIELD instruction 10-97
SET CPU TIMER instruction 10-98
SET FPC instruction 19-43
set prefix (signal-processor order) 4-57
SET PREFIX instruction 10-98
SET PROGRAM MASK instruction 7-175
SET PSW KEY FROM ADDRESS instruction 10-99
SET ROUNDING MODE (SRNM) 19-43
SET SECONDARY ASN instruction 10-99
access registers 5-49
SET SECONDARY ASN WITH INSTANCE instruction 10-99
set state (of TOD clock) 4-40
SET STORAGE KEY EXTENDED instruction 10-105
SET SYSTEM MASK instruction 10-105
SFPC (SET FPC) instruction 19-43
SG (SUBTRACT) binary instruction 7-187
SGF (SUBTRACT) binary instruction 7-187
SGFR (SUBTRACT) binary instruction 7-187
SGR (SUBTRACT) binary instruction 7-187
SH (SUBTRACT HALFWORD) instruction 7-188
shared storage
See storage sharing
shared TOD clock 4-40
SHIFT AND ROUND DECIMAL instruction 8-11
examples A-37
SHIFT LEFT DOUBLE instruction 7-176
example A-29
SHIFT LEFT DOUBLE LOGICAL instruction 7-176
SHIFT LEFT SINGLE instruction 7-177
example A-30
SHIFT LEFT SINGLE LOGICAL instruction 7-178
SHIFT RIGHT DOUBLE instruction 7-178
SHIFT RIGHT DOUBLE LOGICAL instruction 7-178
SHIFT RIGHT SINGLE instruction 7-179
SHIFT RIGHT SINGLE LOGICAL instruction 7-180
shifting
floating-point
See normalization
short binary-floating-point number 19-4
short hexadecimal-floating-point number 18-3
short I/O block 16-23
SHY (SUBTRACT HALFWORD) instruction 7-188
SI instruction format 5-6
SID
See subsystem-identification word
sign bit
binary 7-3
floating-point 18-1
sign codes (decimal) 8-2
signal (I/O) 17-12
clear
See clear signal
halt
See halt signal
reset
See reset signal
SIGNAL PROCESSOR instruction 10-106
orders 4-56
status 4-61
signaling NaN (SNaN) 19-6
signed binary
arithmetic 7-4
comparison 7-5
integer 7-3
tables A-2
significance
loss 18-1
in HFP addition 18-9
mask (in PSW) 4-6
starter (in EDIT) 8-7
significand 19-4
SIGP
See SIGNAL PROCESSOR instruction
SIGP (SIGNAL PROCESSOR) instruction 10-106
single-access reference 5-98
single-path mode 15-3, 15-21
SIY instruction format 5-6
size notation xx
size of address 3-5
controlled by addressing mode 5-8
in CCW 15-28
skip flag in CCW 15-28
effect on data transfer of 15-35

SL (SUBTRACT LOGICAL) instruction 7-189
SLA (SHIFT LEFT SINGLE) instruction 7-177
example A-30
SLAG (SHIFT LEFT SINGLE) instruction 7-177
SLB (SUBTRACT LOGICAL WITH BORROW) instruction 7-189
SLBG (SUBTRACT LOGICAL WITH BORROW) instruction 7-190
SLBGR (SUBTRACT LOGICAL WITH BORROW) instruction 7-189
SLBR (SUBTRACT LOGICAL WITH BORROW) instruction 7-189
SLDA (SHIFT LEFT DOUBLE) instruction 7-176
example A-29
SLDL (SHIFT LEFT DOUBLE LOGICAL) instruction 7-176
SLGR (SUBTRACT LOGICAL) instruction 7-188, 7-189
SLI (suppress-length-indication) flag in CCW 15-28
for immediate operations 15-31
SLL (SHIFT LEFT SINGLE LOGICAL) instruction 7-178
SLLG (SHIFT LEFT SINGLE LOGICAL) instruction 7-178
SLR (SUBTRACT LOGICAL) instruction 7-188
SLY (SUBTRACT LOGICAL) instruction 7-189
SNaN (signaling NaN) 19-6
solicited interruption condition (I/O) 16-3
solid errors 11-5
sorting
extended 1-13
sorting instructions
See also COMPARE AND FORM CODEWORD instruction, UPDATE TREE instruction example A-51
source of interruption
identified by interruption code 6-5
SP (SUBTRACT DECIMAL) instruction 7-12
space-switch event 6-30
control bit
in ASTE 3-27
space-switch-event-control bit
home 3-38
primary 3-36
special-operation exception 6-31
special QNaN 19-6
specification exception 6-33
SPKA (SET PSW KEY FROM ADDRESS) instruction 10-99
SPM (SET PROGRAM MASK) instruction 7-175
SPT (SET CPU TIMER) instruction 10-98
SPX (SET PREFIX) instruction 10-98
SQD (SQUARE ROOT) HFP instruction 18-21
SQDB (SQUARE ROOT) BFP instruction 19-44
SQDBR (SQUARE ROOT) BFP instruction 19-44
SQDR (SQUARE ROOT) HFP instruction 18-21
SQE (SQUARE ROOT) HFP instruction 18-21
SQEB (SQUARE ROOT) BFP instruction 19-44
SQEBR (SQUARE ROOT) BFP instruction 19-44
SQER (SQUARE ROOT) HFP instruction 18-21
SQUARE ROOT BFP instructions 19-44
SQUARE ROOT HFP instructions 18-21
SQXBR (SQUARE ROOT) BFP instruction 19-44
SQXR (SQUARE ROOT) HFP instruction 18-21
SR (SUBTRACT) binary instruction 7-187
SRA (SHIFT RIGHT SINGLE) instruction 7-179
SRAG (SHIFT RIGHT SINGLE) instruction 7-179
SRDA (SHIFT RIGHT DOUBLE) instruction 7-178
SRDL (SHIFT RIGHT DOUBLE LOGICAL) instruction 7-179
SRL (SHIFT RIGHT SINGLE LOGICAL) instruction 7-180
SRLG (SHIFT RIGHT SINGLE LOGICAL) instruction 7-180
SRN (SET ROUNDING MODE) 19-43
SRP (SHIFT AND ROUND DECIMAL) instruction 8-11
examples A-37
SRSD (secondary real-space designation) 3-37
SRST (SEARCH STRING) instruction 7-172
examples A-29
SRSTU (SEARCH STRING UNICODE) instruction 7-173
SRTD (secondary region-table designation) 3-37
SRTL (secondary region-table length) 3-37
SROTO (secondary region-table origin) 3-37
SS instruction format 5-6
SSAIR (SET SECONDARY ASN WITH INSTANCE) instruction 10-99
SSAR (SET SECONDARY ASN) instruction 10-99
access registers 5-49
SSAR-cp or SSAIR-cp (SET SECONDARY ASN (WITH INSTANCE) instruction, to current primary) 10-100
SSAR-ss or SSAIR-ss (SET SECONDARY ASN (WITH INSTANCE) instruction, with space switching) 10-100
SSASTEO (subspace-AST-entry origin) 5-65
SSASTESN (subspace-AST-entry sequence number) 5-66
SSCH (START SUBCHANNEL) instruction 14-14
SSE instruction format 5-6
SSKE (SET STORAGE KEY EXTENDED)
SSTD (secondary segment-table designation) 3-37
SSTL (secondary segment-table length) 3-37
SSTO (secondary segment-table origin) 3-37
ST (STORE) binary instruction 7-180
stack-empty exception 6-34
stack-full exception 6-35
stack-operation exception 6-35
X-38 z/Architecture Principles of Operation
Index

stack-specification exception 6-35
stack-type exception 6-35
stacking process 5-83
stacking PROGRAM CALL 5-70
STAM (STORE ACCESS MULTIPLE) instruction 7-181
STAMY (STORE ACCESS MULTIPLE) instruction 7-181
standalone dump 12-5
standard epoch (for TOD clock) 4-42
STAP (STORE CPU ADDRESS) instruction 10-108
start (CPU)
  function 4-2
  key 12-5
  signal-processor order 4-56
start function (I/O) 13-6, 15-18
  bit in SCSW for 16-12
  initiated by START SUBCHANNEL 14-14
  path management for 15-19
  pending 16-14
START SUBCHANNEL instruction 14-14
  See also start function for I/O
count of in measurement block 17-4
defered condition code for (in SCSW) 16-8
  function initiated by 15-18
  operation-request block (ORB) used by 15-22
state
  CPU
    See CPU state
  TOD-clock 4-40
state entry 5-80
status
  alert 16-16
  device 16-23
    effect of clear function on 15-15
    field-validity flag for (in subchannel logout) 16-34
    with inappropriate bit combination 16-35
  device-status check 16-35
  for SIGNAL PROCESSOR 4-56, 10-106
  initial-status interruption
    See initial-status-interruption control
  intermediate 16-17
  primary 16-18
program
  See PSW
resulting from signal-processor orders 4-61
  secondary 16-18
storing of 4-55
  manual key for 12-5
  subchannel 16-23
status-control field (in SCSW) 16-16
status modifier (device status)
  effect of in command chaining 15-35
  status pending 16-18
status-verification facility 17-23
STC (STORE CHARACTER) instruction 7-181
STCK (STORE CLOCK) instruction 7-182
STCKC (STORE CLOCK COMPARATOR) instruction 10-107
STCKE (STORE CLOCK EXTENDED) instruction 7-183
STCM (STORE CHARACTERS UNDER MASK) instruction 7-181
  examples A-30
STCMH (STORE CHARACTERS UNDER MASK) instruction 7-181
STCMY (STORE CHARACTERS UNDER MASK) instruction 7-181
STCPS (STORE CHANNEL PATH STATUS) instruction 14-16
STCRW (STORE CHANNEL REPORT WORD) instruction 14-17
STCTG (STORE CONTROL) instruction 10-107
STCTL (STORE CONTROL) instruction 10-107
STCY (STORE CHARACTER) instruction 7-181
STD
  See segment-table designation
STD (STORE) floating-point instruction 9-11
STDY (STORE) floating point instruction 9-11
STE (STORE) floating-point instruction 9-11
STCY (STORE) floating point instruction 9-11
STFL (STORE FACILITY LIST) instruction 10-110
STFL facility list 3-61
STFPC (STORE FPC) instruction 19-44
STG (STORE) binary instruction 7-180
STH (STORE HALFWORD) instruction 7-185
STHY (STORE HALFWORD) instruction 7-185
STIDP (STORE CPU ID) instruction 10-108
STL (segment-table length) 3-36
STM (STORE MULTIPLE) instruction 7-185
  example A-30
STMG (STORE MULTIPLE) instruction 7-185
STMH (STORE MULTIPLE HIGH) instruction 7-186
STMY (STORE MULTIPLE) instruction 7-185
STNSM (STORE THEN AND SYSTEM MASK) instruction 10-124
STO (segment-table origin) 3-36
stop
  function 4-2
  key 12-5
  signal-processor order 4-56
stop and store status (signal-processor order) 4-57
stopped (signal-processor status) 4-62
stopped state
  of CPU 4-1
    effect on completion of store operations 5-97
  of TOD clock 4-40
storage 3-1, 3-36
  absolute 3-3
  address wraparound
    See wraparound
storage (continued)
addressing 3-2
See also address
alteration
space-control bit 4-28
alteration manual controls 12-2
alteration PER event 3-36, 4-35
bits for 3-36
mask for 4-28
assigned locations in 3-58
auxiliary 3-2, 3-33
block 3-3
testing for usability of 10-127
buffer (cache) 3-2
clearing of
See clearing operation
concurrency of access for references to 5-99
configuration of 3-3
direct-access 3-2
display 12-2
derror 11-19
indirect 11-20
expanded 2-2
failing address in
See failing-storage address
interlocked update 5-97
interlocks for virtual references 5-90
main 3-2
noninterlocked update of 5-97
nonvolatile 3-2
operand 5-7
reference to (fetch, store, update) 5-96
update reference 5-97
operand consistency 5-98
examples A-47, A-49
prefixing for 3-15
real 3-4
sequence of references to 5-88
for floating-point data 9-2
size
notation for xx
validation of 11-6
virtual 3-33
volatile 3-2
effect of power-on reset on 4-53
storage-access code (in subchannel logout) 16-34
storage-alteration-event bit 4-28
storage-alteration-event-control bit 3-36
home 3-38
primary 3-36
secondary 3-37
storage-area designation
for I/O operations 15-29
for PER events 4-33
storage degradation (machine-check condition) 11-20
storage key 3-8
error in 11-20
sequence of references to 5-95
testing for usability of 10-127
validation of 11-7
storage-key function 1-14
storage-logical-validity bit 11-22
storage protection 3-9
during tracing 4-27
storage-protection-override-control bit 3-10
storage reconfiguration 1-14
storage sharing
by address spaces 3-33
by CPUs and the channel subsystem 3-3
examples A-43
in multiprocessing 4-55
STORE ACCESS MULTIPLE instructions 7-181
STORE binary instructions 7-180
STORE CHANNEL PATH STATUS instruction 14-16
STORE CHANNEL REPORT WORD instruction 14-17
channel-report word (CRW) stored by 17-26
STORE CHARACTER instructions 7-181
STORE CHARACTERS UNDER MASK instruction
examples A-30
STORE CHARACTERS UNDER MASK
instructions 7-181
STORE CLOCK COMPARE instruction 10-107
STORE CLOCK EXTENDED instruction 7-183
STORE CLOCK instruction 7-182
STORE CONTROL instruction 10-107
STORE CPU ADDRESS instruction 10-108
STORE CPU ID instruction 10-108
STORE CPU TIMER instruction 10-109
STORE facility LIST instruction 10-110
STORE floating-point instructions 9-11
STORE FPC instruction 19-44
STORE HALFWORD instructions 7-185
STORE MULTIPLE HIGH instruction 7-186
STORE MULTIPLE instructions
example A-30
STORE MULTIPLE instructions 7-185
STORE PAIR TO QUADWORD instruction 7-186
STORE PREFIX instruction 10-110
STORE REAL ADDRESS instruction 10-111
store reference 5-96
access exceptions for 6-39
STORE REVERSED instructions 7-187
store status 4-55
key 12-5
signal-processor order for 4-57
store-status architectural-mode identification 3-60
store status at address (signal-processor order) 4-57
STORE SUBCHANNEL instruction 14-17
STORE SYSTEM INFORMATION instruction 10-112
STORE THEN AND SYSTEM MASK
instruction 10-124
STORE THEN OR SYSTEM MASK instruction 10-124
store using real address (PER event) 4-35
store-using-real-address-event mask 4-28
STORE USING REAL ADDRESS instruction 10-124
STOSM (STORE THEN OR SYSTEM MASK) instruction 10-124
STPQ (STORE PAIR TO QUADWORD)
instruction 7-186
STPT (STORE CPU TIMER) instruction 10-109
STPX (STORE PREFIX) instruction 10-110
STRAG (STORE REAL ADDRESS) instruction 10-111
streaming-mode control 15-23
caused by clock comparator 4-45
caused by CPU timer 4-47
STRV (STORE REVERSED) instruction 7-187
STRVG (STORE REVERSED) instruction 7-187
STRVH (STORE REVERSED) instruction 7-187
STSI (STORE SYSTEM INFORMATION) instruction 10-112
STURA (STORE USING REAL ADDRESS)
instruction 10-124
STURG (STORE USING REAL ADDRESS)
instruction 10-124
STY (STORE) binary instruction 7-180
SU (SUBTRACT UNNORMALIZED) HFP
instruction 18-22
subchannel 13-2
active allegiance for 15-12
dedicated allegiance for 15-12
effect of I/O-system reset on 17-15
idle 16-13
working allegiance for 15-12
subchannel-active bit 16-15
subchannel addressing 13-5
subchannel control information in SCSW 16-11
subchannel enabled bit in PMCW 15-2
subchannel-information block
See SCHIB
subchannel key 15-22, 16-8
used as access key 3-9
used for IPL 17-18
subchannel key check (in subchannel logout) 16-32
subchannel logout 16-32
subchannel number 13-5
subchannel status 16-23
subchannel-status word
See SCSW
subclass-mask bits
external-interruption 6-10
I/O-interruption
See I/O-interruption subclass mask
machine-check 11-24
subroutine linkage 5-11
subspace-active bit 5-65
subspace-AST-entry origin (SSASTEO) 5-65
subspace-AST-entry sequence number (SSASTESN) 5-66
subspace-group control 3-36
subspace-group-control bit
primary 3-36
secondary 3-37
subspace groups 5-64
introduction to 5-15
subspace-replacement exceptions 6-47
operations 5-68
subsystem-identification word (SID)
subsystem-identification word (SID)
assigned storage locations for 3-61
subsystem-linkage-control bit 5-26, 5-32, 5-33
subsystem reset 4-52
subsystem-identification word (SID) 14-1
SUBTRACT BFP instructions 19-44
SUBTRACT binary instructions 7-187
SUBTRACT DECIMAL instruction 8-12
SUBTRACT HALFWORD instructions 7-188
SUBTRACT LOGICAL instructions 7-188
SUBTRACT LOGICAL WITH BORROW
instructions 7-189
SUBTRACT NORMALIZED
See SUBTRACT BFP instructions
SUBTRACT NORMALIZED HFP instructions 18-22
SUBTRACT UNNORMALIZED HFP instructions 18-22
successful-branching PER event 4-34
mask for 4-28
SUPERVISOR CALL instruction 7-190
supervisor-call interruption 6-47
supervisor state 4-6
support functions (I/O) 17-1
suppress-length-indication flag in CCW
See SLI
suppress-suspended-interruption control (I/O) 15-25, 16-11
used for IPL 17-18
suppression exceptions to 5-23
of instruction execution 5-20
of unit of operation 5-22
suppression on protection 3-12
SUR (SUBTRACT UNNORMALIZED) HFP
instruction 18-22
suspend control 15-22
suspend-control bit 16-8
used for IPL 17-18
suspend flag in CCW 15-28
invalid 16-25
suspend function 13-8
suspended bit (in SCSW) 16-16
suspension of channel-program execution 15-38
effect on DCTI of 15-40
intermediate interruption condition for 16-17
SVC (SUPERVISOR CALL) instruction 7-190
SW (SUBTRACT UNNORMALIZED) HFP
instruction 18-22
swapping
by COMPARE (DOUBLE) AND SWAP
instructions 7-53
by EXCLUSIVE OR instruction 7-119
SWR (SUBTRACT UNNORMALIZED) HFP
instruction 18-22
SX (segment index) 3-34
SXBR (SUBTRACT) BFP instruction 19-44
SXR (SUBTRACT NORMALIZED) HFP
instruction 18-22
SY (SUBTRACT) binary instruction 7-187
symbol translation
use by VTAM 7-80
symbol-translation-option bit 7-73
synchronization
checkpoint 11-3
of CPU timer with TOD clock 4-46
of TOD clocks 4-40, 4-44
synchronization control 15-24
synchronous machine-check-interruption
conditions 11-18
system
manual control of 12-1
organization of 2-1
system check stop 11-11
system damage 11-16
system mask (in PSW) 4-3
validity bit for 11-21
system recovery 11-16
system reset
See reset
I/O
See I/O-system reset
system-reset-clear key 12-5
system-reset-normal key 12-5
System/360 and System/370 I/O interface
See parallel-I/O interface
tables (continued)
DAT
See page table, segment table, region table
entry
See entry table
linkage
See linkage table
page
See page table
region
See region table
segment
See segment table
trace 4-10
translation 3-38
TAI (International Atomic Time) related to UTC 4-42
TAM (TEST ADDRESSING MODE) instruction 7-190
TAR (TEST ACCESS) instruction 10-125
target instruction 7-119
TB (TEST BLOCK) instruction 10-127
TBDR (CONVERT HFP TO BFP) floating-point instruction
TBDR (CONVERT HFP TO BFP) floating-point instruction 9-9
TBEDR (CONVERT HFP TO BFP) floating point instruction 9-9
TCDB (TEST DATA CLASS) BFP instruction 19-45
TCEB (TEST DATA CLASS) BFP instruction 19-45
TCXB (TEST DATA CLASS) BFP instruction 19-45
termination
of I/O operations
See conclusion of I/O operations
of instruction execution 5-21
for exigent machine-check conditions 11-11
of unit of operation 5-22
for exigent machine-check conditions 11-11
termination code (in subchannel logout) 16-34
field-validity flag for 16-34
TEST ACCESS instruction 10-125
TEST ADDRESSING MODE instruction 7-190
TEST AND SET instruction 7-191
TEST BLOCK instruction 10-127
TEST DATA CLASS BFP instructions 19-45
TEST DECIMAL instruction 8-13
test indicator 12-5
TEST PENDING INTERRUPTION instruction 14-18, 14-19
TEST PROTECTION instruction 10-129
TEST SUBCHANNEL instruction 14-20
interruption-response block (IRB) used by 16-6
TEST UNDER MASK HIGH instruction 7-192
TEST UNDER MASK instruction
examples A-31
TEST UNDER MASK instructions 7-191
TEST UNDER MASK LOW instruction 7-192

T (tera) xx
table of powers of 2 G-1
table-type bits 3-39, 3-40
in region-table entry 3-39
in segment-table entry 3-40
tables
ASN
See ASN first table, ASN second table
authority
See authority table
testing for storage-block and storage-key usability 10-127
THDER (CONVERT BFP TO HFP) floating-point instruction 9-8
THDR (CONVERT BFP TO HFP) floating-point instruction 9-8
TIC (transfer in channel) 15-41
invalid sequence of 16-26
time-of-day clock
See TOD clock
timer
See CPU timer
timing
channel-subsystem 17-2
timing facilities 4-40
timing-facility bit (in PMCW) 15-4
timing-facility damage 11-16
for TOD clock 4-41
TLB (translation-lookaside buffer) 3-50
entries 3-50
attachment of 3-50
clearing of 3-53
effect of translation changes on 3-52
usable state 3-51
TM (TEST UNDER MASK) instruction 7-191
examples A-31
TMH (TEST UNDER MASK HIGH) instruction 7-192
TMHH (TEST UNDER MASK) instruction 7-191
TML (TEST UNDER MASK LOW) instruction 7-192
TMLH (TEST UNDER MASK) instruction 7-192
TMLL (TEST UNDER MASK) instruction 7-192
TMY (TEST UNDER MASK) instruction 7-191
TOD clock 4-40
effect of power-on reset on 4-53
effect on clock-comparator interruption 6-11
effect on CPU-timer decrementing 4-46
effect on CPU-timer interruption 6-11
manual control of 4-40, 12-6
unique values of 4-41
validation of 11-10
value in trace entry 4-26
TOD-clock-control-override control 4-40
TOD-clock programmable field 4-42
TOD-clock programmable register 4-42
save areas for 3-63
validity bit for 11-22
TOD-clock-sync-control bit 4-40, 4-44
TOD-clock-synchronization facility 4-44
TP (TEST DECIMAL) instruction 8-13
TPI (TEST PENDING INTERRUPTION) instruction 14-18, 14-19
interuption code stored by 14-19
TPROT (TEST PROTECTION) instruction 10-129
TR (TRANSLATE) instruction 7-192
example A-31
trace 4-10
entries 4-14
entry address 4-14
exceptions 6-47
table address 6-35
TRACE instruction 10-132
trace entry for 4-26
TRACG (TRACE) instruction 10-132
trailer entry 5-80
transfer in channel
See TIC
transferring program control 5-70
TRANSLATE AND TEST instruction 7-193
example A-32
TRANSLATE AND TEST REVERSE instruction 7-194
TRANSLATE EXTENDED instruction 7-195
TRANSLATE instruction 7-192
example A-31
TRANSLATE ONE TO ONE instruction 7-198
TRANSLATE ONE TO TWO instruction 7-198
TRANSLATE TWO TO ONE instruction 7-198
TRANSLATE TWO TO TWO instruction 7-198
translation
address 3-33
See also dynamic address translation
exception identification 3-60
lookaside buffer
See TLB
PC-number 5-31
specification exception 6-36
tables for 3-38
translation modes 3-35
translation parameters 3-35
translation path 3-51
trap control block 10-134
TRAP instruction 10-133
trap save area 10-135
TRAP2 (TRAP) instruction 10-133
TRAP4 (TRAP) instruction 10-133
TRE (TRANSLATE EXTENDED) instruction 7-195
tree structure for sorting 7-205
example A-51
trial execution
for editing instructions and TRANSLATE instruction 5-24
for PER 4-29
trimodal addressing 5-8
TROO (TRANSLATE ONE TO ONE) instruction 7-198
TROT (TRANSLATE ONE TO TWO) instruction 7-198
TRT (TRANSLATE AND TEST) instruction 7-193
example A-32
TRTO (TRANSLATE TWO TO ONE) instruction 7-198
TRTR (TRANSLATE AND TEST REVERSE) instruction 7-194
TRTT (TRANSLATE TWO TO TWO) instruction 7-198
true zero (HFP number) 18-1
TS (TEST AND SET) instruction 7-191
TSCH (TEST SUBCHANNEL) instruction 14-20
two's complement binary notation 7-3
examples A-2
type of PROGRAM CALL 5-35

U
ulp (unit in the last place) 9-3, 19-4, 19-7
underflow
See exponent underflow
unit check (device status)
in establishing dedicated allegiance 15-12
unit of operation 5-21
unlock A-46
example with FIFO queuing A-48
example with LIFO queuing A-47
unnormalized floating-point number 18-3
HFP data only 9-1
unordered (comparison to a NaN) 19-8
unordered comparison 19-23
UNPACK ASCII instruction 7-203
UNPACK instruction 7-202
example A-33
UNPACK UNICODE instruction 7-204
UNPK (UNPACK) instruction 7-202
example A-33
UNPKA (PACK ASCII) instruction 7-203
UNPKU (PACK UNICODE) instruction 7-204
unprivileged instructions 4-6, 7-2
unsigned binary
arithmetic 7-5
integer 7-3
examples A-3
in address generation 5-9
unsolicited interruption condition (I/O) 16-3
unstack-suppression bit 5-78
unstacking process 5-86
update reference 5-97
UPDATE TREE instruction 7-205
example A-51
UPT (UPDATE TREE) instruction 7-205
example A-51
usable ALB entry 5-63
usable TLB entry 3-51
UTC (Coordinated Universal Time) used in TOD
epoch 4-42

V
valid ART-table entry 5-63
valid CBC 11-2
valid floating-point-register numbers 9-2
valid region-table, segment-table, or page-table
entry 3-50
validation 11-5
of registers 11-10
of storage 11-6
of storage key 11-7
of TOD clock 11-10
validity bit for backward stack-entry address 5-79
validity bit for forward-section-header address 5-80
validity bits
in machine-check-interruption code 11-21
in subchannel logout 16-34
variable-length field 3-3
version code 10-108
virtual address 3-4
virtual machine
extensions for 1-14
virtual storage 3-33
VM-data-space facility 1-14
volatile storage 3-2
effect of power-on reset on 4-53
VTAM use of symbol translation 7-80

W
WAIT (SVC)
example of routine to bypass A-46
wait indicator 12-6
wait-state bit
in PSW 4-6
warning (machine-check condition) 11-17
subclass-mask bit for 11-25
word 3-3
word-concurrent storage references 5-99
working allegiance (I/O) 15-12
wraparound
of instruction addresses 5-8
of PER addresses 4-33
of register numbers
for LOAD MULTIPLE instruction 7-129
for STORE MULTIPLE instruction 7-186
of storage addresses 3-6
controlled by addressing mode 3-6
for MOVE INVERSE instruction 7-134
for MOVE LONG EXTENDED instruction 7-139
for MOVE LONG instruction 7-135
for MOVE LONG UNICODE instruction 7-144
of TOD clock 4-40

X
X (EXCLUSIVE OR) instruction 7-118
X field of instruction 5-8
XA (extended architecture)
See 370-XA architecture
XC (EXCLUSIVE OR) instruction 7-118
examples A-19
Z

Z bit (zero condition-code bit) 16-11
as cause of intermediate interruption condition 16-17
z/Architecture architecture
additions to 1-1, 1-7
original highlights of 1-1
ZAP (ZERO AND ADD) instruction 8-13
example A-38
zero
instruction-length code 6-7
negative
See negative zero
normal meaning for byte value xxi
setting floating-point register to 9-11
true (HFP number) 18-1
ZERO AND ADD instruction 8-13
example A-38
zero condition code (Z bit in SCSW) 16-11
zone bits 8-1
moving of 7-148
zoned decimal numbers 8-1
examples A-4
Communicating Your Comments to IBM

z/Architecture
Principles of Operation
Publication No. SA22-7832-03

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+845+432-9405
- If you prefer to send comments electronically, use one of these network IDs:
  - Internet e-mail: mhvrdfs@us.ibm.com

Make sure to include the following in your note:
- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.
Reader's Comments — We'd Like to Hear from You

z/Architecture
Principles of Operation
Publication No. SA22-7832-03

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: ________________________________

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ] As an introduction          [ ] As a text (student)
[ ] As a reference manual       [ ] As a text (instructor)
[ ] For another purpose (explain)

__________________________________________________________________________

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name
Address

Company or Organization

Phone No.