

Alpha Architecture Handbook

Order Number EC–QD2KB–TE


Digital Equipment Corporation
Maynard, Massachusetts
October 1996

While Digital believes the information included in this publication is correct as of the date of publication, it is subject to change without notice.

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Copyright ©1996 Digital Equipment Corporation. All rights reserved.

The following are trademarks of Digital Equipment Corporation: AlphaGeneration, Alpha AXP, AXP, DEC, Digital, OpenVMS, PDP--11, VAX, VAX DOCUMENT, the AlphaGeneration design mark, and the DIGITAL logo.

Cray is a registered trademark of Cray Research, Inc. IBM is a registered trademark of International Business Machines Corporation. UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Ltd. Windows NT is a trademark of Microsoft Corporation.

All other trademarks and registered trademarks are the property of their respective holders.

This document was prepared using VAX DOCUMENT, Version 2.1-1.
## Contents

### Preface

#### Chapter 1 Introduction (I)

1.1 The Alpha Approach to RISC Architecture .................................................. 1-1  
1.2 Data Format Overview .................................................................................. 1-3  
1.3 Instruction Format Overview ....................................................................... 1-4  
1.4 Instruction Overview .................................................................................. 1-5  
1.5 Instruction Set Characteristics .................................................................... 1-6  
1.6 Terminology and Conventions ..................................................................... 1-7  
1.6.1 Numbering ........................................................................................ 1-7  
1.6.2 Security Holes ..................................................................................... 1-7  
1.6.3 UNPREDICTABLE and UNDEFINED ............................................... 1-7  
1.6.4 Ranges and Extents .............................................................................. 1-8  
1.6.5 ALIGNED and UNALIGNED ............................................................. 1-9  
1.6.6 Must Be Zero (MBZ) ........................................................................... 1-9  
1.6.7 Read As Zero (RAZ) ............................................................................ 1-9  
1.6.8 Should Be Zero (SBZ) .......................................................................... 1-9  
1.6.9 Ignore (IGN) ....................................................................................... 1-9  
1.6.10 Implementation Dependent (IMP) ...................................................... 1-9  
1.6.12 Macro Code Example Conventions .................................................... 1-9  

#### Chapter 2 Basic Architecture (I)

2.1 Addressing ......................................................................................... 2-1  
2.2 Data Types ......................................................................................... 2-1  
2.2.1 Byte ................................................................................................ 2-1  
2.2.2 Word ............................................................................................... 2-2  
2.2.3 Longword ........................................................................................ 2-2  
2.2.4 Quadword ........................................................................................ 2-2  
2.2.5 VAX Floating-Point Formats ............................................................. 2-3  
2.2.5.1 F_floating ................................................................................... 2-3  
2.2.5.2 G_floating ................................................................................... 2-5  
2.2.5.3 D_floating ................................................................................... 2-6  
2.2.6 IEEE Floating-Point Formats ............................................................. 2-7  
2.2.6.1 S_Floating .................................................................................. 2-7  
2.2.6.2 T_floating ................................................................................... 2-9  
2.2.6.3 X_Floating .................................................................................. 2-10
### Chapter 3 Instruction Formats (I)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Alpha Registers</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.1 Program Counter</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.2 Integer Registers</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.3 Floating-Point Registers</td>
<td>3-2</td>
</tr>
<tr>
<td>3.1.4 Lock Registers</td>
<td>3-2</td>
</tr>
<tr>
<td>3.1.5 Processor Cycle Counter (PCC) Register</td>
<td>3-2</td>
</tr>
<tr>
<td>3.1.6 Optional Registers</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.6.1 Memory Prefetch Registers</td>
<td>3-3</td>
</tr>
<tr>
<td>3.1.6.2 VAX Compatibility Register</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2 Notation</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2.1 Operand Notation</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2.2 Instruction Operand Notation</td>
<td>3-4</td>
</tr>
<tr>
<td>3.2.3 Operators</td>
<td>3-6</td>
</tr>
<tr>
<td>3.2.4 Notation Conventions</td>
<td>3-9</td>
</tr>
<tr>
<td>3.3 Instruction Formats</td>
<td>3-9</td>
</tr>
<tr>
<td>3.3.1 Memory Instruction Format</td>
<td>3-9</td>
</tr>
<tr>
<td>3.3.1.1 Memory Format Instructions with a Function Code</td>
<td>3-10</td>
</tr>
<tr>
<td>3.3.1.2 Memory Format Jump Instructions</td>
<td>3-10</td>
</tr>
<tr>
<td>3.3.2 Branch Instruction Format</td>
<td>3-11</td>
</tr>
<tr>
<td>3.3.3 Operate Instruction Format</td>
<td>3-11</td>
</tr>
<tr>
<td>3.3.4 Floating-Point Operate Instruction Format</td>
<td>3-12</td>
</tr>
<tr>
<td>3.3.4.1 Floating-Point Convert Instructions</td>
<td>3-13</td>
</tr>
<tr>
<td>3.3.4.2 Floating-Point/Integer Register Moves</td>
<td>3-13</td>
</tr>
<tr>
<td>3.3.5 PALcode Instruction Format</td>
<td>3-13</td>
</tr>
</tbody>
</table>

### Chapter 4 Instruction Descriptions (I)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Instruction Set Overview</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1.1 Subsetting Rules</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.1.1 Floating-Point Subsets</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2 Software Emulation Rules</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.3 Opcode Qualifiers</td>
<td>4-3</td>
</tr>
<tr>
<td>4.2 Memory Integer Load/Store Instructions</td>
<td>4-4</td>
</tr>
<tr>
<td>4.2.1 Load Address</td>
<td>4-5</td>
</tr>
<tr>
<td>4.2.2 Load Memory Data into Integer Register</td>
<td>4-6</td>
</tr>
<tr>
<td>4.2.3 Load Unaligned Memory Data into Integer Register</td>
<td>4-8</td>
</tr>
<tr>
<td>4.2.4 Load Memory Data into Integer Register Locked</td>
<td>4-9</td>
</tr>
<tr>
<td>4.2.5 Store Integer Register Data into Memory Conditional</td>
<td>4-12</td>
</tr>
</tbody>
</table>
4.2.6 Store Integer Register Data into Memory ................................ 4-15
4.2.7 Store Unaligned Integer Register Data into Memory ...................... 4-17
4.3 Control Instructions ....................................................................... 4-18
4.3.1 Conditional Branch ..................................................................... 4-20
4.3.2 Unconditional Branch .................................................................. 4-22
4.3.3 Jumps ......................................................................................... 4-23
4.4 Integer Arithmetic Instructions ............................................... 4-25
4.4.1 Longword Add ............................................................................ 4-26
4.4.2 Scaled Longword Add ................................................................... 4-27
4.4.3 Quadword Add ............................................................................. 4-28
4.4.4 Scaled Quadword Add ................................................................... 4-29
4.4.5 Integer Signed Compare ............................................................. 4-30
4.4.6 Integer Unsigned Compare ......................................................... 4-31
4.4.7 Count Leading Zero ..................................................................... 4-32
4.4.8 Count Population .......................................................................... 4-33
4.4.9 Count Trailing Zero ...................................................................... 4-34
4.4.10 Longword Multiply .................................................................... 4-35
4.4.11 Quadword Multiply ..................................................................... 4-36
4.4.12 Unsigned Quadword Multiply High ............................................. 4-37
4.4.13 Longword Subtract .................................................................... 4-38
4.4.14 Scaled Longword Subtract ......................................................... 4-39
4.4.15 Quadword Subtract ..................................................................... 4-40
4.4.16 Scaled Quadword Subtract ........................................................ 4-41
4.5 Logical and Shift Instructions .................................................... 4-42
4.5.1 Logical Functions ......................................................................... 4-43
4.5.2 Conditional Move Integer .............................................................. 4-44
4.5.3 Shift Logical ................................................................................. 4-46
4.5.4 Shift Arithmetic ............................................................................ 4-47
4.6 Byte-Manipulation Instructions ..................................................... 4-48
4.6.1 Compare Byte .............................................................................. 4-50
4.6.2 Extract Byte ................................................................................ 4-52
4.6.3 Byte Insert .................................................................................. 4-56
4.6.4 Byte Mask .................................................................................. 4-58
4.6.5 Sign Extend .................................................................................. 4-61
4.6.6 Zero Bytes ................................................................................... 4-62
4.7 Floating-Point Instructions ............................................................ 4-63
4.7.1 Floating-Point Single-Precision Operations .................................... 4-63
4.7.2 Floating Subsets and Floating Faults ............................................. 4-63
4.7.3 Definitions ................................................................................... 4-64
4.7.4 Encodings .................................................................................... 4-65
4.7.5 Floating-Point Rounding Modes .................................................. 4-66
4.7.6 Floating-Point Trapping Modes ................................................... 4-68
4.7.6.1 Imprecise/Software Completion Trap Modes ............................ 4-69
4.7.6.2 Invalid Operation (INV) Arithmetic Trap ................................. 4-71
4.7.6.3 Division by Zero (DZE) Arithmetic Trap .................................... 4-72
5.6.1  Alpha Shared Memory Model ................................................ 5-11
5.6.1.1 Architectural Definition of Processor Issue Sequence .............. 5-12
5.6.1.2 Definition of Before and After ........................................ 5-12
5.6.1.3 Definition of Processor Issue Constraints ........................ 5-13
5.6.1.4 Definition of Location Access Constraints ......................... 5-14
5.6.1.5 Definition of Visibility ................................................ 5-14
5.6.1.6 Definition of Storage .................................................. 5-14
5.6.1.7 Definition of Dependence Constraints .............................. 5-15
5.6.1.8 Definition of Load-Locked and Store-Conditional ................. 5-16
5.6.1.9 Timeliness ............................................................. 5-17
5.6.2  Litmus Tests ............................................................. 5-17
5.6.2.1 Litmus Test 1 (Impossible Sequence) ............................. 5-17
5.6.2.2 Litmus Test 2 (Impossible Sequence) ............................. 5-18
5.6.2.3 Litmus Test 3 (Impossible Sequence) ............................. 5-18
5.6.2.4 Litmus Test 4 (Sequence Okay) ..................................... 5-19
5.6.2.5 Litmus Test 5 (Sequence Okay) ..................................... 5-19
5.6.2.6 Litmus Test 6 (Sequence Okay) ..................................... 5-19
5.6.2.7 Litmus Test 7 (Impossible Sequence) ............................. 5-20
5.6.2.8 Litmus Test 8 (Impossible Sequence) ............................. 5-20
5.6.2.9 Litmus Test 9 (Impossible Sequence) ............................. 5-21
5.6.2.10 Litmus Test 10 (Sequence Okay) .................................. 5-21
5.6.2.11 Litmus Test 11 (Impossible Sequence) ............................ 5-21
5.6.3  Implied Barriers ......................................................... 5-22
5.6.4  Implications for Software ............................................... 5-22
5.6.4.1 Single-Processor Data Stream ...................................... 5-22
5.6.4.2 Single-Processor Instruction Stream ................................ 5-22
5.6.4.3 Multiple-Processor Data Stream (Including Single Processor with DMA I/O) ...................................................... 5-23
5.6.4.4 Multiple-Processor Instruction Stream (Including Single Processor with DMA I/O) ...................................................... 5-23
5.6.4.5 Multiple-Processor Context Switch .................................. 5-24
5.6.4.6 Multiple-Processor Send/Receive Interrupt ......................... 5-26
5.6.4.7 Implications for Memory Mapped I/O ............................... 5-27
5.6.4.8 Multiple Processors Writing to a Single I/O Device .............. 5-28
5.6.5  Implications for Hardware ............................................... 5-28
5.7  Arithmetic Traps ............................................................. 5-29

Chapter 6  Common PALcode Architecture (I)

6.1  PALcode ................................................................. 6-1
6.2  PALcode Instructions and Functions .................................... 6-1
6.3  PALcode Environment .................................................... 6-2
6.4  Special Functions Required for PALcode ................................ 6-2
6.5  PALcode Effects on System Code ....................................... 6-3
6.6  PALcode Replacement ..................................................... 6-3
6.7  Required PALcode Instructions ........................................... 6-4
6.7.1  Drain Aborts ............................................................ 6-6
# Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>Instruction Format Overview</td>
<td>1–4</td>
</tr>
<tr>
<td>2–1</td>
<td>Byte Format</td>
<td>2–1</td>
</tr>
<tr>
<td>2–2</td>
<td>Word Format</td>
<td>2–2</td>
</tr>
<tr>
<td>2–3</td>
<td>Longword Format</td>
<td>2–2</td>
</tr>
<tr>
<td>2–4</td>
<td>Quadword Format</td>
<td>2–3</td>
</tr>
<tr>
<td>2–5</td>
<td>F-floating Datum</td>
<td>2–3</td>
</tr>
<tr>
<td>2–6</td>
<td>F-floating Register Format</td>
<td>2–3</td>
</tr>
<tr>
<td>2–7</td>
<td>G-floating Datum</td>
<td>2–5</td>
</tr>
<tr>
<td>2–8</td>
<td>G-floating Format</td>
<td>2–5</td>
</tr>
<tr>
<td>2–9</td>
<td>D-floating Datum</td>
<td>2–6</td>
</tr>
<tr>
<td>2–10</td>
<td>D-floating Register Format</td>
<td>2–6</td>
</tr>
<tr>
<td>2–11</td>
<td>S-floating Datum</td>
<td>2–8</td>
</tr>
<tr>
<td>2–12</td>
<td>S-floating Register Format</td>
<td>2–8</td>
</tr>
<tr>
<td>2–13</td>
<td>T-floating Datum</td>
<td>2–9</td>
</tr>
<tr>
<td>2–14</td>
<td>T-floating Register Format</td>
<td>2–9</td>
</tr>
<tr>
<td>2–15</td>
<td>X_Floating Datum</td>
<td>2–10</td>
</tr>
<tr>
<td>2–16</td>
<td>X_Floating Register Format</td>
<td>2–10</td>
</tr>
<tr>
<td>2–17</td>
<td>X_Floating Big-Endian Datum</td>
<td>2–11</td>
</tr>
<tr>
<td>2–18</td>
<td>X_Floating Big-Endian Register Format</td>
<td>2–12</td>
</tr>
<tr>
<td>2–19</td>
<td>Longword Integer Datum</td>
<td>2–12</td>
</tr>
<tr>
<td>2–20</td>
<td>Longword Integer Floating-Register Format</td>
<td>2–12</td>
</tr>
<tr>
<td>2–21</td>
<td>Quadword Integer Datum</td>
<td>2–13</td>
</tr>
<tr>
<td>2–22</td>
<td>Quadword Integer Floating-Register Format</td>
<td>2–13</td>
</tr>
<tr>
<td>2–23</td>
<td>Little-Endian Byte Addressing</td>
<td>2–14</td>
</tr>
<tr>
<td>2–24</td>
<td>Big-Endian Byte Addressing</td>
<td>2–14</td>
</tr>
<tr>
<td>3–1</td>
<td>Memory Instruction Format</td>
<td>3–10</td>
</tr>
<tr>
<td>3–2</td>
<td>Memory Instruction with Function Code Format</td>
<td>3–10</td>
</tr>
<tr>
<td>3–3</td>
<td>Branch Instruction Format</td>
<td>3–11</td>
</tr>
<tr>
<td>3–4</td>
<td>Operate Instruction Format</td>
<td>3–11</td>
</tr>
<tr>
<td>3–5</td>
<td>Floating-Point Operate Instruction Format</td>
<td>3–12</td>
</tr>
<tr>
<td>3–6</td>
<td>PALcode Instruction Format</td>
<td>3–13</td>
</tr>
<tr>
<td>4–1</td>
<td>Floating-Point Control Register (FPCR) Format</td>
<td>4–74</td>
</tr>
<tr>
<td>4–2</td>
<td>Floating-Point Instruction Function Field</td>
<td>4–79</td>
</tr>
<tr>
<td>8–1</td>
<td>Alpha System Overview</td>
<td>8–1</td>
</tr>
<tr>
<td>B–1</td>
<td>IEEE Floating-Point Control (FP_C) Quadword</td>
<td>B–4</td>
</tr>
<tr>
<td>B–2</td>
<td>IEEE Trap Handling Behavior</td>
<td>B–6</td>
</tr>
</tbody>
</table>
Tables

2–1 F_floating Load Exponent Mapping (MAP_F) .................................................. 2–4
2–2 S_floating Load Exponent Mapping (MAP_S) ............................................... 2–8
3–1 Operand Notation ...................................................................................... 3–3
3–2 Operand Value Notation ............................................................................. 3–4
3–3 Expression Operand Notation .................................................................... 3–4
3–4 Operators .................................................................................................... 3–6
4–1 Opcode Qualifiers ..................................................................................... 4–3
4–2 Memory Integer Load/Store Instructions .................................................... 4–4
4–3 Control Instructions Summary .................................................................... 4–19
4–4 Jump Instructions Branch Prediction ............................................................ 4–24
4–5 Integer Arithmetic Instructions Summary .................................................. 4–25
4–6 Logical and Shift Instructions Summary ..................................................... 4–42
4–7 Byte-Manipulation Instructions Summary .................................................. 4–48
4–8 Floating-Point Control Register (FPCR) Bit Descriptions ......................... 4–74
4–9 IEEE Floating-Point Function Field Bit Summary ....................................... 4–80
4–10 VAX Floating-Point Function Field Bit Summary ...................................... 4–81
4–11 Memory Format Floating-Point Instructions Summary ............................ 4–84
4–12 Floating-Point Branch Instructions Summary .......................................... 4–93
4–13 Floating-Point Operate Instructions Summary ......................................... 4–96
4–14 Miscellaneous Instructions Summary ...................................................... 4–131
4–15 VAX Compatibility Instructions Summary .............................................. 4–149
5–1 Processor Issue Constraints ...................................................................... 5–13
6–1 PALcode Instructions that Require Recognition ........................................... 6–4
6–2 Required PALcode Instructions ..................................................................... 6–5
9–1 Unprivileged OpenVMS Alpha PALcode Instruction Summary .................. 9–1
9–2 Privileged OpenVMS Alpha PALcode Instructions Summary .................... 9–7
10–1 Unprivileged Digital UNIX PALcode Instruction Summary ..................... 10–1
10–2 Privileged Digital UNIX PALcode Instruction Summary .......................... 10–2
11–1 Unprivileged Windows NT Alpha PALcode Instruction Summary ............. 11–1
11–2 Privileged Windows NT Alpha PALcode Instruction Summary .................. 11–2
A–1 Decodable Pseudo-Operations (Stylized Code Forms) ............................... A–16
B–1 Floating-Point Control (FP_C) Quadword Bit Summary ............................. B–4
B–2 IEEE Floating-Point Trap Handling ............................................................. B–7
B–3 IEEE Standard Charts ............................................................................... B–12
C–1 Instruction Format and Opcode Notation ................................................... C–1
C–2 Common Architecture Instructions ............................................................. C–1
C–3 IEEE Floating-Point Instruction Function Codes ........................................ C–7
C–4 VAX Floating-Point Instruction Function Codes ......................................... C–9
C–5 Independent Floating-Point Instruction Function Codes ............................ C–9
C–6 Opcode Summary ....................................................................................... C–12
C–7 Key to Opcode Summary (Table C–6) .......................................................... C–12
C–8 Common Architecture Opcodes in Numerical Order .................................. C–13
C–9 OpenVMS Alpha Unprivileged PALcode Instructions ............................... C–17
Preface

Chapters 1 through 8 and appendixes A through D of this book are directly derived from the Alpha System Reference Manual and passed engineering change orders (ECOs) that have been applied. It is an accurate representation of the described parts of the Alpha architecture. Additions and modifications to those described parts of the Alpha architecture include new and modified instructions, significant changes to Chapter 5, and the addition of two tables in a new Appendix D.

New Instructions

Three extensions have been added to the Alpha architecture with sets of new instructions. The extensions and their mnemonics are:

<table>
<thead>
<tr>
<th>Extension</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte/word</td>
<td>BWX</td>
</tr>
<tr>
<td>Count</td>
<td>CIX</td>
</tr>
<tr>
<td>Multimedia</td>
<td>MAX</td>
</tr>
</tbody>
</table>

Two instructions (AMASK and IMPLVER) have been added that can determine whether one or more extensions are supported.

Two instructions (ECB and WH64) have been added to provide greater efficiency in data-cache operations.

The new instructions are listed in the following table.

<table>
<thead>
<tr>
<th>Description</th>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte/word</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte/word extension (BWX):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDBU</td>
<td></td>
<td>Load zero-extended byte from memory to register</td>
</tr>
<tr>
<td>LDWU</td>
<td></td>
<td>Load zero-extended word from memory to register</td>
</tr>
<tr>
<td>SEXTB</td>
<td></td>
<td>Sign extend byte</td>
</tr>
<tr>
<td>SEXTW</td>
<td></td>
<td>Sign extend word</td>
</tr>
<tr>
<td>STB</td>
<td></td>
<td>Store byte</td>
</tr>
<tr>
<td>STW</td>
<td></td>
<td>Store word</td>
</tr>
</tbody>
</table>

**Count extension (CIX):**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CTLZ</td>
<td>Count leading zero</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Mnemonic</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>Count population</td>
<td>CTPOP</td>
<td>Count trailing zero</td>
</tr>
<tr>
<td>Floating-point to integer register move, $S_{floating}$</td>
<td>FTOIS</td>
<td>Floating-point to integer register move, $T_{floating}$</td>
</tr>
<tr>
<td>Integer to floating-point register move, $F_{floating}$</td>
<td>ITOFF</td>
<td>Integer to floating-point register move, $S_{floating}$</td>
</tr>
<tr>
<td>Integer to floating-point register move, $T_{floating}$</td>
<td>ITOFS</td>
<td>Integer to floating-point register move, $T_{floating}$</td>
</tr>
<tr>
<td>Square root $F_{floating}$</td>
<td>SQRTF</td>
<td>Square root $G_{floating}$</td>
</tr>
<tr>
<td>Square root $S_{floating}$</td>
<td>SQRTS</td>
<td>Square root $T_{floating}$</td>
</tr>
<tr>
<td>Square root $T_{floating}$</td>
<td>SQRTT</td>
<td></td>
</tr>
</tbody>
</table>

**Multimedia extension (MAX):**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Signed Byte Maximum</td>
<td>MAXSB8</td>
</tr>
<tr>
<td>Vector Signed Word Maximum</td>
<td>MAXSW4</td>
</tr>
<tr>
<td>Vector Unsigned Byte Maximum</td>
<td>MAXUB8</td>
</tr>
<tr>
<td>Vector Unsigned Word Maximum</td>
<td>MAXUW4</td>
</tr>
<tr>
<td>Vector Signed Byte Minimum</td>
<td>MINSB8</td>
</tr>
<tr>
<td>Vector Signed Word Minimum</td>
<td>MINSW4</td>
</tr>
<tr>
<td>Vector Unsigned Byte Minimum</td>
<td>MINUB8</td>
</tr>
<tr>
<td>Vector Unsigned Word Minimum</td>
<td>MINUW4</td>
</tr>
<tr>
<td>Pixel Error</td>
<td>PERR</td>
</tr>
<tr>
<td>Pack longwords to bytes</td>
<td>PKLB</td>
</tr>
<tr>
<td>Pack words to bytes</td>
<td>PKWB</td>
</tr>
<tr>
<td>Unpack bytes to longwords</td>
<td>UNPKBL</td>
</tr>
<tr>
<td>Unpack bytes to words</td>
<td>UNPKBW</td>
</tr>
</tbody>
</table>

**Test for extension support:**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture mask</td>
<td>AMASK</td>
</tr>
<tr>
<td>Implementation version</td>
<td>IMPLVER</td>
</tr>
</tbody>
</table>

**Enhance data-cache efficiency:**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evict cache block</td>
<td>ECB</td>
</tr>
<tr>
<td>Write hint—64 bytes</td>
<td>WH64</td>
</tr>
</tbody>
</table>
Modified Instructions:
The following instructions have been modified:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDx_L</td>
<td>Semantic changes</td>
</tr>
<tr>
<td>STx_C</td>
<td>Semantic changes</td>
</tr>
<tr>
<td>WMB</td>
<td>Tighten the definition</td>
</tr>
</tbody>
</table>

Appendix D
Appendix D provides two sections. The first, Processor Type assignments, contains descriptions for the current processors. The second, Architecture Mask and Implementation Version Assignments, lists values for the AMASK and IMPLVER instructions that specify support for the architecture extensions and code tuning.

For More Information
A more extensive description of the operating system and console interface ECO changes and additions, including the new instructions and support for extended virtual address space, is located at:

Chapter 1
Introduction (I)

Alpha is a 64-bit load/store RISC architecture that is designed with particular emphasis on the three elements that most affect performance: clock speed, multiple instruction issue, and multiple processors.

The Alpha architects examined and analyzed current and theoretical RISC architecture design elements and developed high-performance alternatives for the Alpha architecture. The architects adopted only those design elements that appeared valuable for a projected 25-year design horizon. Thus, Alpha becomes the first 21st century computer architecture.

The Alpha architecture is designed to avoid bias toward any particular operating system or programming language. Alpha supports the OpenVMS Alpha, Digital UNIX, and Windows NT Alpha operating systems and supports simple software migration for applications that run on those operating systems.

This manual describes in detail how Alpha is designed to be the leadership 64-bit architecture of the computer industry.

1.1 The Alpha Approach to RISC Architecture

**Alpha Is a True 64-Bit Architecture**

Alpha was designed as a 64-bit architecture. All registers are 64 bits in length and all operations are performed between 64-bit registers. It is not a 32-bit architecture that was later expanded to 64 bits.

**Alpha Is Designed for Very High-Speed Implementations**

The instructions are very simple. All instructions are 32 bits in length. Memory operations are either loads or stores. All data manipulation is done between registers.

The Alpha architecture facilitates pipelining multiple instances of the same operations because there are no special registers and no condition codes.

The instructions interact with each other only by one instruction writing a register or memory and another instruction reading from the same place. That makes it particularly easy to build implementations that issue multiple instructions every CPU cycle. (The first implementation issues two instructions per cycle.)

Alpha makes it easy to maintain binary compatibility across multiple implementations and easy to maintain full speed on multiple-issue implementations. For example, there are no implementation-specific pipeline timing hazards, no load-delay slots, and no branch-delay slots.
The Alpha Approach to Byte Manipulation
The Alpha architecture reads and writes bytes between registers and memory with the LDBU and STB instructions. (Alpha also supports word read/writes with the LDWU and STW instructions.)

Byte shifting and masking is performed with normal 64-bit register-to-register instructions, crafted to keep instruction sequences short.

The Alpha Approach to Arithmetic Traps
Alpha lets the software implementor determine the precision of arithmetic traps. With the Alpha architecture, arithmetic traps (such as overflow and underflow) are imprecise—they can be delivered an arbitrary number of instructions after the instruction that triggered the trap. Also, traps from many different instructions can be reported at once. That makes implementations that use pipelining and multiple issue substantially easier to build.

However, if precise arithmetic exceptions are desired, trap barrier instructions can be explicitly inserted in the program to force traps to be delivered at specific points.

The Alpha Approach to Multiprocessor Shared Memory
As viewed from a second processor (including an I/O device), a sequence of reads and writes issued by one processor may be arbitrarily reordered by an implementation. This allows implementations to use multibank caches, bypassed write buffers, write merging, pipelined writes with retry on error, and so forth. If strict ordering between two accesses must be maintained, explicit memory barrier instructions can be inserted in the program.

The basic multiprocessor interlocking primitive is a RISC-style load_locked, modify, store_conditional sequence. If the sequence runs without interrupt, exception, or an interfering write from another processor, then the conditional store succeeds. Otherwise, the store fails and the program eventually must branch back and retry the sequence. This style of interlocking scales well with very fast caches, and makes Alpha an especially attractive architecture for building multiple-processor systems.

Alpha Instructions Include Hints for Achieving Higher Speed
A number of Alpha instructions include hints for implementations, all aimed at achieving higher speed.

- Calculated jump instructions have a target hint that can allow much faster subroutine calls and returns.
- There are prefetching hints for the memory system that can allow much higher cache hit rates.
- There are granularity hints for the virtual-address mapping that can allow much more effective use of translation lookaside buffers for large contiguous structures.

PALcode—Alpha’s Very Flexible Privileged Software Library
A Privileged Architecture Library (PALcode) is a set of subroutines that are specific to a particular Alpha operating system implementation. These subroutines provide operating-system primitives for context switching, interrupts, exceptions,
and memory management. PALcode is similar to the BIOS libraries that are provided in personal computers.

PALcode subroutines are invoked by implementation hardware or by software CALL_PAL instructions.

PALcode is written in standard machine code with some implementation-specific extensions to provide access to low-level hardware.

PALcode lets Alpha implementations run the full OpenVMS Alpha, Digital UNIX, and Windows NT Alpha operating systems. PALcode can provide this functionality with little overhead. For example, the OpenVMS Alpha PALcode instructions let Alpha run OpenVMS with little more hardware than that found on a conventional RISC machine: the PAL mode bit itself, plus 4 extra protection bits in each translation buffer entry.

Other versions of PALcode can be developed for real-time, teaching, and other applications.

PALcode makes Alpha an especially attractive architecture for multiple operating systems.

Alpha and Programming Languages

Alpha is an attractive architecture for compiling a large variety of programming languages. Alpha has been carefully designed to avoid bias toward one or two programming languages. For example:

- Alpha does not contain a subroutine call instruction that moves a register window by a fixed amount. Thus, Alpha is a good match for programming languages with many parameters and programming languages with no parameters.

- Alpha does not contain a global integer overflow enable bit. Such a bit would need to be changed at every subroutine boundary when a FORTRAN program calls a C program.

1.2 Data Format Overview

Alpha is a load/store RISC architecture with the following data characteristics:

- All operations are done between 64-bit registers.
- Memory is accessed via 64-bit virtual byte addresses, using the little-endian or, optionally, the big-endian byte numbering convention.
- There are 32 integer registers and 32 floating-point registers.
- Longword (32-bit) and quadword (64-bit) integers are supported.
- Five floating-point data types are supported:
  - VAX F_floating (32-bit)
  - VAX G_floating (64-bit)
  - IEEE single (32-bit)
— IEEE double (64-bit)
— IEEE extended (128-bit)

1.3 Instruction Format Overview

As shown in Figure 1-1, Alpha instructions are all 32 bits in length. As represented in Figure 1-1, there are four major instruction format classes that contain 0, 1, 2, or 3 register fields. All formats have a 6-bit opcode.

Figure 1–1: Instruction Format Overview

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Number</th>
<th>PALcode Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>RA</td>
<td>Branch Format</td>
</tr>
<tr>
<td>Opcode</td>
<td>RA RB</td>
<td>Memory Format</td>
</tr>
<tr>
<td>Opcode</td>
<td>RA RB</td>
<td>Operate Format</td>
</tr>
</tbody>
</table>

- **PALcode instructions** specify, in the function code field, one of a few dozen complex operations to be performed.

- **Conditional branch instructions** test register Ra and specify a signed 21-bit PC-relative longword target displacement. Subroutine calls put the return address in register Ra.

- **Load and store instructions** move bytes, words, longwords or quadwords between register Ra and memory, using Rb plus a signed 16-bit displacement as the memory address.

- **Operate instructions** for floating-point and integer operations are both represented in Figure 1–1 by the operate format illustration and are as follows:
  
  — Word and byte sign-extension operators.

  — Floating-point operations use Ra and Rb as source registers, and write the result in register Rc. There is an 11-bit extended opcode in the function field.

  — Integer operations use Ra and Rb or an 8-bit literal as the source operand, and write the result in register Rc.

  Integer operate instructions can use the Rb field and part of the function field to specify an 8-bit literal. There is a 7-bit extended opcode in the function field.
1.4 Instruction Overview

PALcode Instructions
As described above, a Privileged Architecture Library (PALcode) is a set of subroutines that is specific to a particular Alpha operating-system implementation. These subroutines can be invoked by hardware or by software CALL_PAL instructions, which use the function field to vector to the specified subroutine.

Branch Instructions
Conditional branch instructions can test a register for positive/negative or for zero/nonzero. They can also test integer registers for even/odd.

Unconditional branch instructions can write a return address into a register.

There is also a calculated jump instruction that branches to an arbitrary 64-bit address in a register.

Load/Store Instructions
Load and store instructions move 8-bit, 16-bit, 32-bit or 64-bit aligned quantities from and to memory. Memory addresses are flat 64-bit virtual addresses, with no segmentation.

The VAX floating-point load/store instructions swap words to give a consistent register format for floating-point operations.

A 32-bit integer datum is placed in a register in a canonical form that makes 33 copies of the high bit of the datum. A 32-bit floating-point datum is placed in a register in a canonical form that extends the exponent by 3 bits and extends the fraction with 29 low-order zeros. The 32-bit operates preserve these canonical forms.

Compilers, as directed by user declarations, can generate any mixture of 32-bit and 64-bit operations. The Alpha architecture has no 32/64 mode bit.

Integer Operate Instructions
The integer operate instructions manipulate full 64-bit values, and include the usual assortment of arithmetic, compare, logical, and shift instructions.

There are just three 32-bit integer operates: add, subtract, and multiply. They differ from their 64-bit counterparts only in overflow detection and in producing 32-bit canonical results.

There is no integer divide instruction.

The Alpha architecture also supports the following additional operations:

• Scaled add/subtract instructions for quick subscript calculation
• 128-bit multiply for division by a constant, and multiprecision arithmetic
• Conditional move instructions for avoiding branch instructions
• An extensive set of in-register byte and word manipulation instructions
Integer overflow trap enable is encoded in the function field of each instruction, rather than kept in a global state bit. Thus, for example, both ADDQ/V and ADDQ opcodes exist for specifying 64-bit ADD with and without overflow checking. That makes it easier to pipeline implementations.

Floating-Point Operate Instructions
The floating-point operate instructions include four complete sets of VAX and IEEE arithmetic instructions, plus instructions for performing conversions between floating-point and integer quantities.

In addition to the operations found in conventional RISC architectures, Alpha includes conditional move instructions for avoiding branches and merge sign/exponent instructions for simple field manipulation.

The arithmetic trap enables and rounding mode are encoded in the function field of each instruction, rather than kept in global state bits. That makes it easier to pipeline implementations.

1.5 Instruction Set Characteristics

Alpha instruction set characteristics are as follows:

• All instructions are 32 bits long and have a regular format.

• There are 32 integer registers (R0 through R31), each 64 bits wide. R31 reads as zero, and writes to R31 are ignored.

  All integer data manipulation is between integer registers, with up to two variable register source operands (one may be an 8-bit literal), and one register destination operand.

  Instructions can move data in an integer register file to a floating-point register file. The instructions do not interpret bits in the register files and do not access memory.

• There are 32 floating-point registers (F0 through F31), each 64 bits wide. F31 reads as zero, and writes to F31 are ignored.

  All floating-point data manipulation is between floating-point registers, with up to two register source operands and one register destination operand.

  Instructions can move data in an floating-point register file to an integer register file. The instructions do not interpret bits in the register files and do not access memory.

• All memory reference instructions are of the load/store type that move data between registers and memory.

• There are no branch condition codes. Branch instructions test an integer or floating-point register value, which may be the result of a previous compare.

• Integer and logical instructions operate on quadwords.
Floating-point instructions operate on G_floating, F_floating, IEEE double, and IEEE single operands. D_floating "format compatibility," in which binary files of D_floating numbers may be processed, but without the last 3 bits of fraction precision, is also provided.

A minimal number of VAX compatibility instructions are included.

1.6 Terminology and Conventions

The following sections describe the terminology and conventions used in this book.

1.6.1 Numbering

All numbers are decimal unless otherwise indicated. Where there is ambiguity, numbers other than decimal are indicated with the name of the base in subscript form, for example, 10_{16}.

1.6.2 Security Holes

A security hole is an error of commission, omission, or oversight in a system that allows protection mechanisms to be bypassed.

Security holes exist when unprivileged software (that is, software running outside of kernel mode) can:

- Affect the operation of another process without authorization from the operating system;
- Amplify its privilege without authorization from the operating system; or
- Communicate with another process, either overtly or covertly, without authorization from the operating system.

The Alpha architecture has been designed to contain no architectural security holes. Hardware (processors, buses, controllers, and so on) and software should likewise be designed to avoid security holes.

1.6.3 UNPREDICTABLE and UNDEFINED

The terms UNPREDICTABLE and UNDEFINED are used throughout this book. Their meanings are quite different and must be carefully distinguished.

In particular, only privileged software (software running in kernel mode) can trigger UNDEFINED operations. Unprivileged software cannot trigger UNDEFINED operations. However, either privileged or unprivileged software can trigger UNPREDICTABLE results or occurrences.

UNPREDICTABLE results or occurrences do not disrupt the basic operation of the processor; it continues to execute instructions in its normal manner. In contrast, UNDEFINED operation can halt the processor or cause it to lose information.

The terms UNPREDICTABLE and UNDEFINED can be further described as follows:
UNPREDICTABLE

- Results or occurrences specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.

- An UNPREDICTABLE result may acquire an arbitrary value subject to a few constraints. Such a result may be an arbitrary function of the input operands or of any state information that is accessible to the process in its current access mode. UNPREDICTABLE results may be unchanged from their previous values. Operations that produce UNPREDICTABLE results may also produce exceptions.

- An occurrence specified as UNPREDICTABLE may happen or not based on an arbitrary choice function. The choice function is subject to the same constraints as are UNPREDICTABLE results and, in particular, must not constitute a security hole.

Specifically, UNPREDICTABLE results must not depend upon, or be a function of, the contents of memory locations or registers which are inaccessible to the current process in the current access mode.

Also, operations that may produce UNPREDICTABLE results must not:

- Write or modify the contents of memory locations or registers to which the current process in the current access mode does not have access, or
- Halt or hang the system or any of its components.

For example, a security hole would exist if some UNPREDICTABLE result depended on the value of a register in another process, on the contents of processor temporary registers left behind by some previously running process, or on a sequence of actions of different processes.

UNDEFINED

- Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing, to stopping system operation.

- UNDEFINED operations may halt the processor or cause it to lose information. However, UNDEFINED operations must not cause the processor to hang, that is, reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions.

1.6.4 Ranges and Extents

Ranges are specified by a pair of numbers separated by a “..” and are inclusive. For example, a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.
Extents are specified by a pair of numbers in angle brackets separated by a colon and are inclusive. For example, bits <7:3> specify an extent of bits including bits 7, 6, 5, 4, and 3.

1.6.5 ALIGNED and UNALIGNED

In this document the terms ALIGNED and NATURALLY ALIGNED are used interchangeably to refer to data objects that are powers of two in size. An aligned datum of size \(2^N\) is stored in memory at a byte address that is a multiple of \(2^N\), that is, one that has \(N\) low-order zeros. Thus, an aligned 64-byte stack frame has a memory address that is a multiple of 64.

If a datum of size \(2^N\) is stored at a byte address that is not a multiple of \(2^N\), it is called UNALIGNED.

1.6.6 Must Be Zero (MBZ)

Fields specified as Must be Zero (MBZ) must never be filled by software with a non-zero value. These fields may be used at some future time. If the processor encounters a non-zero value in a field specified as MBZ, an Illegal Operand exception occurs.

1.6.7 Read As Zero (RAZ)

Fields specified as Read as Zero (RAZ) return a zero when read.

1.6.8 Should Be Zero (SBZ)

Fields specified as Should be Zero (SBZ) should be filled by software with a zero value. Non-zero values in SBZ fields produce UNPREDICTABLE results and may produce extraneous instruction-issue delays.

1.6.9 Ignore (IGN)

Fields specified as Ignore (IGN) are ignored when written.

1.6.10 Implementation Dependent (IMP)

Fields specified as Implementation Dependent (IMP) may be used for implementation-specific purposes. Each implementation must document fully the behavior of all fields marked as IMP by the Alpha specification.

1.6.11 Figure Drawing Conventions

Figures that depict registers or memory follow the convention that increasing addresses run right to left and top to bottom.

1.6.12 Macro Code Example Conventions

All instructions in macro code examples are either listed in Chapter 4 or OpenVMS AXP Software II–A, Chapter 2, or are stylized code forms found in Appendix A.
2.1 Addressing

The basic addressable unit in the Alpha architecture is the 8-bit byte. Virtual addresses are 64 bits long. An implementation may support a smaller virtual address space. The minimum virtual address size is 43 bits.

Virtual addresses as seen by the program are translated into physical memory addresses by the memory management mechanism.

Although the data types in Section 2.2 are described in terms of little-endian byte addressing, implementations may also include big-endian addressing support, as described in Section 2.3. All current implementations have some big-endian support.

2.2 Data Types

Following are descriptions of the Alpha architecture data types.

2.2.1 Byte

A byte is 8 contiguous bits starting on an addressable byte boundary. The bits are numbered from right to left, 0 through 7, as shown in Figure 2–1.

Figure 2–1: Byte Format

```
7 0
:A
```

A byte is specified by its address A. A byte is an 8-bit value. The byte is only supported in Alpha by the load, store, sign-extend, extract, mask, insert, and zap instructions.
2.2.2 Word

A word is 2 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 15, as shown in Figure 2–2.

Figure 2–2: Word Format

```
 15 0
  
  :A
```

A word is specified by its address, the address of the byte containing bit 0.

A word is a 16-bit value. The word is only supported in Alpha by the load, store, sign-extend, extract, mask, and insert instructions.

2.2.3 Longword

A longword is 4 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 31, as shown in Figure 2–3.

Figure 2–3: Longword Format

```
 31 0
  
  :A
```

A longword is specified by its address A, the address of the byte containing bit 0. A longword is a 32-bit value.

When interpreted arithmetically, a longword is a two's-complement integer with bits of increasing significance from 0 through 30. Bit 31 is the sign bit. The longword is only supported in Alpha by sign-extended load and store instructions and by longword arithmetic instructions.

Note:

Alpha implementations will impose a significant performance penalty when accessing longword operands that are not naturally aligned. (A naturally aligned longword has zero as the low-order two bits of its address.)

2.2.4 Quadword

A quadword is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are numbered from right to left, 0 through 63, as shown in Figure 2–4.
A quadword is specified by its address A, the address of the byte containing bit 0. A quadword is a 64-bit value. When interpreted arithmetically, a quadword is either a two's-complement integer with bits of increasing significance from 0 through 62 and bit 63 as the sign bit, or an unsigned integer with bits of increasing significance from 0 through 63.

Note:
Alpha implementations will impose a significant performance penalty when accessing quadword operands that are not naturally aligned. (A naturally aligned quadword has zero as the low-order three bits of its address.)

2.2.5 VAX Floating-Point Formats
VAX floating-point numbers are stored in one set of formats in memory and in a second set of formats in registers. The floating-point load and store instructions convert between these formats purely by rearranging bits; no rounding or range-checking is done by the load and store instructions.

2.2.5.1 F_floating
An F_floating datum is 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–5.

Figure 2–5: F_floating Datum

An F_floating operand occupies 64 bits in a floating register, left-justified in the 64-bit register, as shown in Figure 2–6.

Figure 2–6: F_floating Register Format
The F_floating load instruction reorders bits on the way in from memory, expands the exponent from 8 to 11 bits, and sets the low-order fraction bits to zero. This produces in the register an equivalent G_floating number suitable for either F_floating or G_floating operations. The mapping from 8-bit memory-format exponents to 11-bit register-format exponents is shown in Table 2–1.

Table 2–1: F_floating Load Exponent Mapping (MAP_F)

<table>
<thead>
<tr>
<th>Memory &lt;14:7&gt;</th>
<th>Register &lt;62:52&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1111111</td>
<td>1 000 1111111</td>
</tr>
<tr>
<td>1 xxxxxxx</td>
<td>1 000 xxxxxxx</td>
</tr>
<tr>
<td>0 xxxxxxx</td>
<td>0 111 xxxxxxx</td>
</tr>
<tr>
<td>0 0000000</td>
<td>0 000 0000000</td>
</tr>
</tbody>
</table>

This mapping preserves both normal values and exceptional values.

The F_floating store instruction reorders register bits on the way to memory and does no checking of the low-order fraction bits. Register bits <61:59> and <28:0> are ignored by the store instruction.

An F_floating datum is specified by its address A, the address of the byte containing bit 0. The memory form of an F_floating datum is sign magnitude with bit 15 the sign bit, bits <14:7> an excess-128 binary exponent, and bits <6:0> and <31:16> a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance are from 16 through 31 and 0 through 6. The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the F_floating datum has a value of 0.

If the result of a VAX floating-point format instruction has a value of zero, the instruction always produces a datum with a sign bit of 0, an exponent of 0, and all fraction bits of 0. Exponent values of 1..255 indicate true binary exponents of -127..127. An exponent value of 0, together with a sign bit of 1, is taken as a reserved operand. Floating-point instructions processing a reserved operand take an arithmetic exception. The value of an F_floating datum is in the approximate range 0.29*10**–38 through 1.7*10**38. The precision of an F_floating datum is approximately one part in 2**23, typically 7 decimal digits. See Section 4.7.

Note:

Alpha implementations will impose a significant performance penalty when accessing F_floating operands that are not naturally aligned. (A naturally aligned F_floating datum has zero as the low-order two bits of its address.)
2.2.5.2 G_floating

A G_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–7.

Figure 2–7: G_floating Datum

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>14</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction Midh</td>
<td>S</td>
<td>Exp.</td>
<td>Frac.Hi</td>
<td>:A</td>
<td></td>
</tr>
<tr>
<td>Fraction Lo</td>
<td>Fraction Midl</td>
<td>:A+4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A G_floating operand occupies 64 bits in a floating register, arranged as shown in Figure 2–8.

Figure 2–8: G_floating Format

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>32</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exp.</td>
<td>Fraction Hi</td>
<td></td>
<td>Fraction Lo</td>
<td>:Fx</td>
<td></td>
</tr>
</tbody>
</table>

A G_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a G_floating datum is sign magnitude with bit 15 the sign bit, bits <14:4> an excess-1024 binary exponent, and bits <3:0> and <63:16> a normalized 53-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance are from 48 through 63, 32 through 47, 16 through 31, and 0 through 3. The 11-bit exponent field encodes the values 0 through 2047. An exponent value of 0, together with a sign bit of 0, is taken to indicate that the G_floating datum has a value of 0.

If the result of a floating-point instruction has a value of zero, the instruction always produces a datum with a sign bit of 0, an exponent of 0, and all fraction bits of 0. Exponent values of 1..2047 indicate true binary exponents of –1023..1023. An exponent value of 0, together with a sign bit of 1, is taken as a reserved operand. Floating-point instructions processing a reserved operand take a user-visible arithmetic exception. The value of a G_floating datum is in the approximate range 0.56*10**–308 through 0.9*10**308. The precision of a G_floating datum is approximately one part in 2**52, typically 15 decimal digits. See Section 4.7.

Note:

Alpha implementations will impose a significant performance penalty when accessing G_floating operands that are not naturally aligned. (A naturally aligned G_floating datum has zero as the low-order three bits of its address.)
2.2.5.3 D_floating

A D_floating datum in memory is 8 contiguous bytes starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–9.

Figure 2–9: D_floating Datum

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction Midh</td>
<td>S</td>
<td>Exp.</td>
<td>Frac.Hi</td>
<td>:A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fraction Lo</td>
<td>Fraction Midl</td>
<td>:A+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A D_floating operand occupies 64 bits in a floating register, arranged as shown in Figure 2–10.

Figure 2–10: D_floating Register Format

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>55</th>
<th>54</th>
<th>48</th>
<th>47</th>
<th>32</th>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exp.</td>
<td>Frac.Hi</td>
<td>Fraction Midh</td>
<td>Fraction Midl</td>
<td>Fraction Lo</td>
<td>:Fx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The reordering of bits required for a D_floating load or store are identical to those required for a G_floating load or store. The G_floating load and store instructions are therefore used for loading or storing D_floating data.

A D_floating datum is specified by its address A, the address of the byte containing bit 0. The memory form of a D_floating datum is identical to an F_floating datum except for 32 additional low significance fraction bits. Within the fraction, bits of increasing significance are from 48 through 63, 32 through 47, 16 through 31, and 0 through 6. The exponent conventions and approximate range of values is the same for D_floating as F_floating. The precision of a D_floating datum is approximately one part in 2**55, typically 16 decimal digits.

Notes:

- D_floating is not a fully supported data type; no D_floating arithmetic operations are provided in the architecture. For backward compatibility, exact D_floating arithmetic may be provided via software emulation. D_floating “format compatibility” in which binary files of D_floating numbers may be processed, but without the last 3 bits of fraction precision, can be obtained via conversions to G_floating, G arithmetic operations, then conversion back to D_floating.

- Alpha implementations will impose a significant performance penalty on access to D_floating operands that are not naturally aligned. (A naturally aligned D_floating datum has zero as the low-order three bits of its address.)
2.2.6 IEEE Floating-Point Formats

The IEEE standard for binary floating-point arithmetic, ANSI/IEEE 754-1985, defines four floating-point formats in two groups, basic and extended, each having two widths, single and double. The Alpha architecture supports the basic single and double formats, with the basic double format serving as the extended single format. The values representable within a format are specified by using three integer parameters:

1. \( P \) — the number of fraction bits
2. \( E_{\text{max}} \) — the maximum exponent
3. \( E_{\text{min}} \) — the minimum exponent

Within each format, only the following entities are permitted:

1. Numbers of the form \((-1)^S \times 2^E \times b(0).b(1)b(2)\ldots b(P-1)\) where:
   a. \( S = 0 \text{ or } 1 \)
   b. \( E = \) any integer between \( E_{\text{min}} \) and \( E_{\text{max}} \), inclusive
   c. \( b(n) = 0 \text{ or } 1 \)
2. Two infinities — positive and negative
3. At least one Signaling NaN
4. At least one Quiet NaN

NaN is an acronym for Not-a-Number. A NaN is an IEEE floating-point bit pattern that represents something other than a number. NaNs come in two forms: Signaling NaNs and Quiet NaNs. Signaling NaNs are used to provide values for uninitialized variables and for arithmetic enhancements. Quiet NaNs provide retrospective diagnostic information regarding previous invalid or unavailable data and results. Signaling NaNs signal an invalid operation when they are an operand to an arithmetic instruction, and may generate an arithmetic exception. Quiet NaNs propagate through almost every operation without generating an arithmetic exception.

Arithmetic with the infinities is handled as if the operands were of arbitrarily large magnitude. Negative infinity is less than every finite number; positive infinity is greater than every finite number.

2.2.6.1 S_Floating

An IEEE single-precision, or S_floating, datum occupies 4 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 31, as shown in Figure 2–11.
An $S_{\text{floating}}$ operand occupies 64 bits in a floating register, left-justified in the 64-bit register, as shown in Figure 2–12.

The $S_{\text{floating}}$ load instruction reorders bits on the way in from memory, expanding the exponent from 8 to 11 bits, and sets the low-order fraction bits to zero. This produces in the register an equivalent $T_{\text{floating}}$ number, suitable for either $S_{\text{floating}}$ or $T_{\text{floating}}$ operations. The mapping from 8-bit memory-format exponents to 11-bit register-format exponents is shown in Table 2–2.

This mapping preserves both normal values and exceptional values. Note that the mapping for all 1’s differs from that of $F_{\text{floating}}$ load, since for $S_{\text{floating}}$ all 1’s is an exceptional value and for $F_{\text{floating}}$ all 1’s is a normal value.

The $S_{\text{floating}}$ store instruction reorders register bits on the way to memory and does no checking of the low-order fraction bits. Register bits $<61:59>$ and $<28:0>$ are ignored by the store instruction. The $S_{\text{floating}}$ load instruction does no checking of the input.

The $S_{\text{floating}}$ store instruction does no checking of the data; the preceding operation should have specified an $S_{\text{floating}}$ result.

An $S_{\text{floating}}$ datum is specified by its address $A$, the address of the byte containing bit 0. The memory form of an $S_{\text{floating}}$ datum is sign magnitude with bit 31 the sign bit, bits $<30:23>$ an excess-127 binary exponent, and bits $<22:0>$ a 23-bit fraction.
The value (V) of an S_floating number is inferred from its constituent sign (S), exponent (E), and fraction (F) fields as follows:

1. If E=255 and F<>0, then V is NaN, regardless of S.
2. If E=255 and F=0, then V = (–1)**S x Infinity.
3. If 0 < E < 255, then V = (–1)**S x 2**(E–127) x (1.F).
4. If E=0 and F<>0, then V = (–1)**S x 2**(-126) x (0.F).
5. If E=0 and F=0, then V = (–1)**S x 0 (zero).

Floating-point operations on S_floating numbers may take an arithmetic exception for a variety of reasons, including invalid operations, overflow, underflow, division by zero, and inexact results.

Note:

Alpha implementations will impose a significant performance penalty when accessing S_floating operands that are not naturally aligned. (A naturally aligned S_floating datum has zero as the low-order two bits of its address.)

### 2.2.6.2 T_floating

An IEEE double-precision, or T_floating, datum occupies 8 contiguous bytes in memory starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 63, as shown in Figure 2–13.

**Figure 2–13: T_floating Datum**

<table>
<thead>
<tr>
<th>31 30</th>
<th>20 19</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exponent</td>
<td>Fraction Lo :A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fraction Hi :A+4</td>
</tr>
</tbody>
</table>

A T_floating operand occupies 64 bits in a floating register, arranged as shown in Figure 2–14.

**Figure 2–14: T_floating Register Format**

<table>
<thead>
<tr>
<th>63 62</th>
<th>52 51</th>
<th>32 31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exp.</td>
<td>Fraction Hi</td>
<td>Fraction Lo :Fx</td>
</tr>
</tbody>
</table>

The T_floating load instruction performs no bit reordering on input, nor does it perform checking of the input data.

The T_floating store instruction performs no bit reordering on output. This instruction does no checking of the data; the preceding operation should have specified a T_floating result.
A T_floating datum is specified by its address A, the address of the byte containing bit 0. The form of a T_floating datum is sign magnitude with bit 63 the sign bit, bits <62:52> an excess-1023 binary exponent, and bits <51:0> a 52-bit fraction.

The value (V) of a T_floating number is inferred from its constituent sign (S), exponent (E), and fraction (F) fields as follows:

1. If E=2047 and F<>0, then V is NaN, regardless of S.
2. If E=2047 and F=0, then V = (-1)**S x Infinity.
3. If 0 < E < 2047, then V = (-1)**S x 2**(E-1023) x (1.F).
4. If E=0 and F<>0, then V = (-1)**S x 2**(-1022) x (0.F).
5. If E=0 and F=0, then V = (-1)**S x 0 (zero).

Floating-point operations on T_floating numbers may take an arithmetic exception for a variety of reasons, including invalid operations, overflow, underflow, division by zero, and inexact results.

Note:

Alpha implementations will impose a significant performance penalty when accessing T_floating operands that are not naturally aligned. (A naturally aligned T_floating datum has zero as the low-order three bits of its address.)

2.2.6.3 X_Floating

Support for 128-bit IEEE extended-precision (X_float) floating-point is initially provided entirely through software. This section is included to preserve the intended consistency of implementation with other IEEE floating-point data types, should the X_float data type be supported in future hardware.

An IEEE extended-precision, or X_floating, datum occupies 16 contiguous bytes in memory, starting on an arbitrary byte boundary. The bits are labeled from right to left, 0 through 127, as shown in Figure 2–15.

Figure 2–15: X_Floating Datum

An X_floating datum occupies two consecutive even/odd floating-point registers (such as F4/F5), as shown in Figure 2–16.
An X_floating datum is specified by its address A, the address of the byte containing bit 0. The form of an X_floating datum is sign magnitude with bit 127 the sign bit, bits <126:112> an excess-16383 binary exponent, and bits <111:0> a 112-bit fraction. The value (V) of an X_floating number is inferred from its constituent sign (S), exponent (E) and fraction (F) fields as follows:

1. If E = 32767 and F ≠ 0, then V is a NaN, regardless of S.
2. If E = 32767 and F = 0, then V = (-1)**S x Infinity.
3. If 0 < E < 32767, then V = (-1)**S x 2**(E-16383) x (1.F).
4. If E = 0 and F ≠ 0, then V = (-1)**S x 2**(-16382) x (0.F).
5. If E = 0 and F = 0, then V = (-1)**S x 0 (zero).

Note:

Alpha implementations will impose a significant performance penalty when accessing X_floating operands that are not naturally aligned. (A naturally aligned X_floating datum has zero as the low-order four bits of its address.)

X_Floating Big-Endian Formats
Section 2.3 describes Alpha support for big-endian data types. It is intended that software or hardware implementation for a big-endian X_float data type comply with that support and have the following formats.
2.2.7 Longword Integer Format in Floating-Point Unit

A longword integer operand occupies 32 bits in memory, arranged as shown in Figure 2–19.

Figure 2–19: Longword Integer Datum

A longword integer operand occupies 64 bits in a floating register, arranged as shown in Figure 2–20.

Figure 2–20: Longword Integer Floating-Register Format

There is no explicit longword load or store instruction; the S_floating load/store instructions are used to move longword data into or out of the floating registers. The register bits <61:59> are set by the S_floating load exponent mapping. They are ignored by S_floating store. They are also ignored in operands of a longword integer operate instruction, and they are set to 000 in the result of a longword operate instruction.

The register format bit <62> “I” in Figure 2–20 is part of the Integer field in Figure 2–19 and represents the high-order bit of that field.

Note:

Alpha implementations will impose a significant performance penalty when accessing longwords that are not naturally aligned. (A naturally aligned longword datum has zero as the low-order two bits of its address.)
2.2.8 Quadword Integer Format in Floating-Point Unit

A quadword integer operand occupies 64 bits in memory, arranged as shown in Figure 2–21.

Figure 2–21: Quadword Integer Datum

A quadword integer operand occupies 64 bits in a floating register, arranged as shown in Figure 2–22.

Figure 2–22: Quadword Integer Floating-Register Format

There is no explicit quadword load or store instruction; the T_floating load/store instructions are used to move quadword data into or out of the floating registers.

The T_floating load instruction performs no bit reordering on input. The T_floating store instruction performs no bit reordering on output. This instruction does no checking of the data; when used to store quadwords, the preceding operation should have specified a quadword result.

Note:

Alpha implementations will impose a significant performance penalty when accessing quadwords that are not naturally aligned. (A naturally aligned quadword datum has zero as the low-order three bits of its address.)

2.2.9 Data Types with No Hardware Support

The following VAX data types are not directly supported in Alpha hardware.

- Octaword
- H_floating
- D_floating (except load/store and convert to/from G_floating)
- Variable-Length Bit Field
- Character String
- Trailing Numeric String
2.3 Big-endian Addressing Support

Alpha implementations may include optional big-endian addressing support.

In a little-endian machine, the bytes within a quadword are numbered right to left:

Figure 2–23:  Little-Endian Byte Addressing

```
7  6  5  4  3  2  1  0
```

In a big-endian machine, they are numbered left to right:

Figure 2–24:  Big-Endian Byte Addressing

```
0  1  2  3  4  5  6  7
```

Bit numbering within bytes is not affected by the byte numbering convention (big-endian or little-endian).

The format for the X_float big-endian data type is shown in Section 2.2.6.3.

The byte numbering convention does not matter when accessing complete aligned quadwords in memory. However, the numbering convention does matter when accessing smaller or unaligned quantities, or when manipulating data in registers, as follows:

- A quadword load or store of data at location 0 moves the same eight bytes under both numbering conventions. However, a longword load or store of data at location 4 must move the leftmost half of a quadword under the little-endian convention, and the rightmost half under the big-endian convention. Thus, to support both conventions, the convention being used must be known and it must affect longword load/store operations.

- A byte extract of byte 5 from a quadword of data into the low byte of a register requires a right shift of 5 bytes under the little-endian convention, but a right shift of 2 bytes under the big-endian convention.

- Manipulating data in a register is almost the same for both conventions. In both, integer and floating-point data have their sign bits in the leftmost byte and their least significant bit in the rightmost byte, so the same integer and
floating-point instructions are used unchanged for both conventions. Big-endian character strings have their most significant character on the left, while little-endian strings have their most significant character on the right.

- The compare byte (CMPBGE) instruction is neutral about direction, doing eight byte compares in parallel. However, following the CMPBGE instruction, the code is different that examines the byte mask to determine which string is larger, depending on whether the rightmost or leftmost unequal byte is used. Thus, compilers must be instructed to generate somewhat different code sequences for the two conventions.

Implementations that include big-endian support must supply all of the following features:

- A means at boot time to choose the byte numbering convention. The implementation is not required to support dynamically changing the convention during program execution. The chosen convention applies to all code executed, both operating-system and user.

- If the big-endian convention is chosen, the longword-length load/store instructions (LDF, LDL, LDL_L, LDS, STF, STL, STL_C, STS) invert bit va<2> (bit 2 of the virtual address). This has the effect of accessing the half of a quadword other than the half that would be accessed under the little-endian convention.

- If the big-endian convention is chosen, the byte manipulation instructions (EXTxx, INSxx, MSKxx) invert bits Rbv<2:0>. This has the effect of changing a shift of 5 bytes into a shift of 2 bytes, for example.

The instruction stream is always considered to be little-endian, and is independent of the chosen byte numbering convention. Compilers, linkers, and debuggers must be aware of this when accessing an instruction stream using data-stream load/store instructions. Thus, the rightmost instruction in a quadword is always executed first and always has the instruction-stream address 0 MOD 8. The same bytes accessed by a longword load/store instruction have data-stream address 0 MOD 8 under the little-endian convention, and 4 MOD 8 under the big-endian convention.

Using either byte numbering convention, it is sometimes necessary to access data that originated on a machine that used the other convention. When this occurs, it is often necessary to swap the bytes within a datum. See Appendix A, Byte Swap, for a suggested code sequence.
3.1 Alpha Registers

Each Alpha processor has a set of registers that hold the current processor state. If an Alpha system contains multiple Alpha processors, there are multiple per-processor sets of these registers.

3.1.1 Program Counter

The Program Counter (PC) is a special register that addresses the instruction stream. As each instruction is decoded, the PC is advanced to the next sequential instruction. This is referred to as the updated PC. Any instruction that uses the value of the PC will use the updated PC. The PC includes only bits <63:2> with bits <1:0> treated as RAZ/IGN. This quantity is a longword-aligned byte address. The PC is an implied operand on conditional branch and subroutine jump instructions. The PC is not accessible as an integer register.

3.1.2 Integer Registers

There are 32 integer registers (R0 through R31), each 64 bits wide.

Register R31 is assigned special meaning by the Alpha architecture. When R31 is specified as a register source operand, a zero-valued operand is supplied.

For all cases except the Unconditional Branch and Jump instructions, results of an instruction that specifies R31 as a destination operand are discarded. Also, it is UNPREDICTABLE whether the other destination operands (implicit and explicit) are changed by the instruction. It is implementation dependent to what extent the instruction is actually executed once it has been fetched. An exception is never signaled for a load that specifies R31 as a destination operation. For all other operations, it is UNPREDICTABLE whether exceptions are signaled during the execution of such an instruction. Note, however, that exceptions associated with the instruction fetch of such an instruction are always signaled.

There are some interesting cases involving R31 as a destination:

- **STx_C R31,disp(Rb)**
  
  Although this might seem like a good way to zero out a shared location and reset the lock_flag, this instruction causes the lock_flag and virtual location \{Rbv + SEXT(disp)\} to become UNPREDICTABLE.

- **LDx_L R31,disp(Rb)**
  
  This instruction produces no useful result since it causes both lock_flag and locked_physical_address to become UNPREDICTABLE.
Unconditional Branch (BR and BSR) and Jump (JMP, JSR, RET, and JSR_COROUTINE) instructions, when R31 is specified as the Ra operand, execute normally and update the PC with the target virtual address. Of course, no PC value can be saved in R31.

3.1.3 Floating-Point Registers
There are 32 floating-point registers (F0 through F31), each 64 bits wide.

When F31 is specified as a register source operand, a true zero-valued operand is supplied. See Section 4.7.3 for a definition of true zero.

Results of an instruction that specifies F31 as a destination operand are discarded and it is UNPREDICTABLE whether the other destination operands (implicit and explicit) are changed by the instruction. In this case, it is implementation-dependent to what extent the instruction is actually executed once it has been fetched. An exception is never signaled for a load that specifies F31 as a destination operation. For all other operations, it is UNPREDICTABLE whether exceptions are signaled during the execution of such an instruction. Note, however, that exceptions associated with the instruction fetch of such an instruction are always signaled.

A floating-point instruction that operates on single-precision data reads all bits <63:0> of the source floating-point register. A floating-point instruction that produces a single-precision result writes all bits <63:0> of the destination floating-point register.

3.1.4 Lock Registers
There are two per-processor registers associated with the LDx.L and STx.C instructions, the lock_flag and the locked_physical_address register. The use of these registers is described in Section 4.2.

3.1.5 Processor Cycle Counter (PCC) Register
The PCC register consists of two 32-bit fields. The low-order 32 bits (PCC<31:0>) are an unsigned, wrapping counter, PCC_CNT. The high-order 32 bits (PCC<63:32>), PCC_OFF, are operating system dependent in their implementation.

PCC_CNT is the base clock register for measuring time intervals, and is suitable for timing intervals on the order of nanoseconds.

PCC_CNT increments once per N CPU cycles, where N is an implementation-specific integer in the range 1..16. The cycle counter frequency is the number of times the processor cycle counter gets incremented per second. The integer count wraps to 0 from a count of FFFF FFFF16. The counter wraps no more frequently than 1.5 times the implementation's interval clock interrupt period (which is two thirds of the interval clock interrupt frequency), which guarantees that an interrupt occurs before PCC_CNT overflows twice.

PCC_OFF need not contain a value related to time and could contain all zeros in a simple implementation. However, if PCC_OFF is used to calculate a per-process or per-thread cycle count, it must contain a value that, when added to PCC_CNT, returns the total PCC register count for that process or thread, modulo 2**32.
Implementation Note:

OpenVMS Alpha and Digital UNIX supply a per-process value in PCC_OFF. PCC is required on all implementations. It is required for every processor, and each processor on a multiprocessor system has its own private, independent PCC. The PCC is read by the RPCC instruction. See Section 4.11.8.

3.1.6 Optional Registers

Some Alpha implementations may include optional memory prefetch or VAX compatibility processor registers.

3.1.6.1 Memory Prefetch Registers

If the prefetch instructions FETCH and FETCH_M are implemented, an implementation will include two sets of state prefetch registers used by those instructions. The use of these registers is described in Section 4.11. These registers are not directly accessible by software and are listed for completeness.

3.1.6.2 VAX Compatibility Register

The VAX compatibility instructions RC and RS include the intr_flag register, as described in Section 4.12.

3.2 Notation

The notation used to describe the operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax.

3.2.1 Operand Notation

Tables 3–1, 3–2, and 3–3 list the notation for the operands, the operand values, and the other expression operands.

Table 3–1: Operand Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ra</td>
<td>An integer register operand in the Ra field of the instruction.</td>
</tr>
<tr>
<td>Rb</td>
<td>An integer register operand in the Rb field of the instruction.</td>
</tr>
<tr>
<td>#b</td>
<td>An integer literal operand in the Rb field of the instruction.</td>
</tr>
<tr>
<td>Rc</td>
<td>An integer register operand in the Rc field of the instruction.</td>
</tr>
<tr>
<td>Fa</td>
<td>A floating-point register operand in the Ra field of the instruction.</td>
</tr>
<tr>
<td>Fb</td>
<td>A floating-point register operand in the Rb field of the instruction.</td>
</tr>
<tr>
<td>Fc</td>
<td>A floating-point register operand in the Rc field of the instruction.</td>
</tr>
</tbody>
</table>
Table 3–2: Operand Value Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rav</td>
<td>The value of the Ra operand. This is the contents of register Ra.</td>
</tr>
<tr>
<td>Rbv</td>
<td>The value of the Rb operand. This could be the contents of register Rb, or a zero-extended 8-bit literal in the case of an Operate format instruction.</td>
</tr>
<tr>
<td>Fav</td>
<td>The value of the floating point Fa operand. This is the contents of register Fa.</td>
</tr>
<tr>
<td>Fbv</td>
<td>The value of the floating point Fb operand. This is the contents of register Fb.</td>
</tr>
</tbody>
</table>

Table 3–3: Expression Operand Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPR_x</td>
<td>Contents of Internal Processor Register x</td>
</tr>
<tr>
<td>IPR_SP[mode]</td>
<td>Contents of the per-mode stack pointer selected by mode</td>
</tr>
<tr>
<td>PC</td>
<td>Updated PC value</td>
</tr>
<tr>
<td>Rn</td>
<td>Contents of integer register n</td>
</tr>
<tr>
<td>Fn</td>
<td>Contents of floating-point register n</td>
</tr>
<tr>
<td>X[m]</td>
<td>Element m of array X</td>
</tr>
</tbody>
</table>

3.2.2 Instruction Operand Notation

The notation used to describe instruction operands follows from the operand specifier notation used in the VAX Architecture Standard. Instruction operands are described as follows:

<name>.<access type><data type>

<name>

Specifies the instruction field (Ra, Rb, Rc, or disp) and register type of the operand (integer or floating). It can be one of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>disp</td>
<td>The displacement field of the instruction.</td>
</tr>
<tr>
<td>fnc</td>
<td>The PAL code function field of the instruction.</td>
</tr>
<tr>
<td>Ra</td>
<td>An integer register operand in the Ra field of the instruction.</td>
</tr>
<tr>
<td>Rb</td>
<td>An integer register operand in the Rb field of the instruction.</td>
</tr>
<tr>
<td>#b</td>
<td>An integer literal operand in the Rb field of the instruction.</td>
</tr>
<tr>
<td>Rc</td>
<td>An integer register operand in the Rc field of the instruction.</td>
</tr>
<tr>
<td>Fa</td>
<td>A floating-point register operand in the Ra field of the instruction.</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>Fb</td>
<td>A floating-point register operand in the Rb field of the instruction.</td>
</tr>
<tr>
<td>Fc</td>
<td>A floating-point register operand in the Rc field of the instruction.</td>
</tr>
</tbody>
</table>

<access type>

Is a letter denoting the operand access type:

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>The operand is used in an address calculation to form an effective address. The data type code that follows indicates the units of addressability (or scale factor) applied to this operand when the instruction is decoded. For example: “.al” means scale by 4 (longwords) to get byte units (used in branch displacements); “.ab” means the operand is already in byte units (used in load/store instructions).</td>
</tr>
<tr>
<td>i</td>
<td>The operand is an immediate literal in the instruction.</td>
</tr>
<tr>
<td>r</td>
<td>The operand is read only.</td>
</tr>
<tr>
<td>m</td>
<td>The operand is both read and written.</td>
</tr>
<tr>
<td>w</td>
<td>The operand is write only.</td>
</tr>
</tbody>
</table>

<data type>

Is a letter denoting the data type of the operand:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Byte</td>
</tr>
<tr>
<td>f</td>
<td>F_floating</td>
</tr>
<tr>
<td>g</td>
<td>G_floating</td>
</tr>
<tr>
<td>l</td>
<td>Longword</td>
</tr>
<tr>
<td>q</td>
<td>Quadword</td>
</tr>
<tr>
<td>s</td>
<td>IEEE single floating (S_floating)</td>
</tr>
<tr>
<td>t</td>
<td>IEEE double floating (T_floating)</td>
</tr>
<tr>
<td>w</td>
<td>Word</td>
</tr>
<tr>
<td>x</td>
<td>The data type is specified by the instruction</td>
</tr>
</tbody>
</table>
3.2.3 Operators

Table 3–4 describes the operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Comment delimiter</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Signed multiplication</td>
</tr>
<tr>
<td>*U</td>
<td>Unsigned multiplication</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation (left argument raised to right argument)</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>←</td>
<td>Replacement</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>Indicates explicit operator precedence</td>
</tr>
<tr>
<td>(x)</td>
<td>Contents of memory location whose address is x</td>
</tr>
<tr>
<td>x&lt;m:n&gt;</td>
<td>Contents of bit field of x defined by bits n through m</td>
</tr>
<tr>
<td>x&lt;m&gt;</td>
<td>M'th bit of x</td>
</tr>
<tr>
<td>ACCESS(x,y)</td>
<td>Accessibility of the location whose address is x using the access mode y. Returns a Boolean value TRUE if the address is accessible, else FALSE.</td>
</tr>
<tr>
<td>AND</td>
<td>Logical product</td>
</tr>
<tr>
<td>ARITH_RIGHT_SHIFT(x,y)</td>
<td>Arithmetic right shift of first operand by the second operand. Y is an unsigned shift value. Bit 63, the sign bit, is copied into vacated bit positions and shifted out bits are discarded.</td>
</tr>
<tr>
<td>BYTE_ZAP(x,y)</td>
<td>X is a quadword, y is an 8-bit vector in which each bit corresponds to a byte of the result. The y bit to x byte correspondence is y&lt;1&gt; ↔ x&lt;8n+7:8n&gt;. This correspondence also exists between y and the result. For each bit of y from n = 0 to 7, if y &lt;n&gt; is 0 then byte &lt;n&gt; of x is copied to byte &lt;n&gt; of result, and if y &lt;n&gt; is 1 then byte &lt;n&gt; of result is forced to all zeros.</td>
</tr>
</tbody>
</table>
Table 3–4 (Cont.): Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE</td>
<td>The CASE construct selects one of several actions based on the value of its argument. The form of a case is:</td>
</tr>
<tr>
<td></td>
<td>CASE argument OF</td>
</tr>
<tr>
<td></td>
<td>argvalue1: action_1</td>
</tr>
<tr>
<td></td>
<td>argvalue2: action_2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>argvaluen: action_n</td>
</tr>
<tr>
<td></td>
<td>[otherwise: default_action]</td>
</tr>
<tr>
<td></td>
<td>ENDCASE</td>
</tr>
<tr>
<td></td>
<td>If the value of argument is argvalue1 then action_1 is executed; if argument = argvalue2, then action_2 is executed, and so forth.</td>
</tr>
<tr>
<td></td>
<td>Once a single action is executed, the code stream breaks to the ENDCASE (there is an implicit break as in Pascal). Each action may nonetheless be a sequence of pseudocode operations, one operation per line.</td>
</tr>
<tr>
<td></td>
<td>Optionally, the last argvalue may be the atom ‘otherwise’. The associated default action will be taken if none of the other argvalues match the argument.</td>
</tr>
<tr>
<td>DIV</td>
<td>Integer division (truncates)</td>
</tr>
<tr>
<td>LEFT_SHIFT(x,y)</td>
<td>Logical left shift of first operand by the second operand.</td>
</tr>
<tr>
<td></td>
<td>Y is an unsigned shift value. Zeros are moved into the vacated bit positions, and shifted out bits are discarded.</td>
</tr>
<tr>
<td>LOAD_LOCKED</td>
<td>The processor records the target physical address in a per-processor locked_physical_address register and sets the per-processor lock_flag.</td>
</tr>
<tr>
<td>lg</td>
<td>Log to the base 2</td>
</tr>
<tr>
<td>MAP_x</td>
<td>F_float or S_float memory-to-register exponent mapping function.</td>
</tr>
<tr>
<td>MAXS(x,y)</td>
<td>Returns the larger of x and y, with x and y interpreted as signed integers</td>
</tr>
<tr>
<td>MAXU(x,y)</td>
<td>Returns the larger of x and y, with x and y interpreted as unsigned integers</td>
</tr>
<tr>
<td>MINS(x,y)</td>
<td>Returns the smaller of x and y, with x and y interpreted as signed integers</td>
</tr>
<tr>
<td>MINU(x,y)</td>
<td>Returns the smaller of x and y, with x and y interpreted as unsigned integers</td>
</tr>
<tr>
<td>x MOD y</td>
<td>x modulo y</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical (ones) complement</td>
</tr>
<tr>
<td>OR</td>
<td>Logical sum</td>
</tr>
</tbody>
</table>
### Table 3–4 (Cont.): Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHYSICAL_ADDRESS</td>
<td>Translation of a virtual address</td>
</tr>
<tr>
<td>PRIORITY_ENCODE</td>
<td>Returns the bit position of most significant set bit, interpreting its argument as a positive integer ((=\text{int}(\lg(x)))). For example: priority_encode(255) = 7</td>
</tr>
</tbody>
</table>

#### Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LT</td>
<td>Less than signed</td>
</tr>
<tr>
<td>LTU</td>
<td>Less than unsigned</td>
</tr>
<tr>
<td>LE</td>
<td>Less or equal signed</td>
</tr>
<tr>
<td>LEU</td>
<td>Less or equal unsigned</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal signed and unsigned</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal signed and unsigned</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal signed</td>
</tr>
<tr>
<td>GEU</td>
<td>Greater or equal unsigned</td>
</tr>
<tr>
<td>GT</td>
<td>Greater signed</td>
</tr>
<tr>
<td>GTU</td>
<td>Greater unsigned</td>
</tr>
<tr>
<td>LBC</td>
<td>Low bit clear</td>
</tr>
<tr>
<td>LBS</td>
<td>Low bit set</td>
</tr>
</tbody>
</table>

| RIGHT_SHIFT(x,y)          | Logical right shift of first operand by the second operand. Y is an unsigned shift value. Zeros are moved into vacated bit positions, and shifted out bits are discarded. |
| SEXT(x)                   | X is sign-extended to the required size. |
| STORE_CONDITIONAL         | If the lock_flag is set, then do the indicated store and clear the lock_flag. |
| TEST(x,cond)              | The contents of register x are tested for branch condition (cond) true. TEST returns a Boolean value TRUE if x bears the specified relation to 0, else FALSE is returned. Integer and floating test conditions are drawn from the preceding list of relational operators. |
| XOR                      | Logical difference             |
| ZEXT(x)                   | X is zero-extended to the required size. |
3.2.4 Notation Conventions

The following conventions are used:

1. Only operands that appear on the left side of a replacement operator are modified.
2. No operator precedence is assumed other than that replacement (←) has the lowest precedence. Explicit precedence is indicated by the use of “{}”.
3. All arithmetic, logical, and relational operators are defined in the context of their operands. For example, “+” applied to G_floating operands means a G_floating add, whereas “+” applied to quadword operands is an integer add. Similarly, “LT” is a G_floating comparison when applied to G_floating operands and an integer comparison when applied to quadword operands.

3.3 Instruction Formats

There are five basic Alpha instruction formats:

- Memory
- Branch
- Operate
- Floating-point Operate
- PAL code

All instruction formats are 32 bits long with a 6-bit major opcode field in bits <31:26> of the instruction.

Any unused register field (Ra, Rb, Fa, Fb) of an instruction must be set to a value of 31.

Software Note:

There are several instructions, each formatted as a memory instruction, that do not use the Ra and/or Rb fields. These instructions are: Memory Barrier, Fetch, Fetch_M, Read Process Cycle Counter, Read and Clear, Read and Set, and Trap Barrier.

3.3.1 Memory Instruction Format

The Memory format is used to transfer data between registers and memory, to load an effective address, and for subroutine jumps. It has the format shown in Figure 3–1.
A Memory format instruction contains a 6-bit opcode field, two 5-bit register address fields, Ra and Rb, and a 16-bit signed displacement field.

The displacement field is a byte offset. It is sign-extended and added to the contents of register Rb to form a virtual address. Overflow is ignored in this calculation.

The virtual address is used as a memory load/store address or a result value, depending on the specific instruction. The virtual address (va) is computed as follows for all memory format instructions except the load address high (LDAH):

$$ va \leftarrow \{Rbv + \text{SEXT}(\text{Memory\_disp})\} $$

For LDAH the virtual address (va) is computed as follows:

$$ va \leftarrow \{Rbv + \text{SEXT}(\text{Memory\_disp} \times 65536)\} $$

### 3.3.1.1 Memory Format Instructions with a Function Code

Memory format instructions with a function code replace the memory displacement field in the memory instruction format with a function code that designates a set of miscellaneous instructions. The format is shown in Figure 3–2.

### 3.3.1.2 Memory Format Jump Instructions

For computed branch instructions (CALL, RET, JMP, JSR_COROUTINE) the displacement field is used to provide branch-prediction hints as described in Section 4.3.
3.3.2 Branch Instruction Format

The Branch format is used for conditional branch instructions and for PC-relative subroutine jumps. It has the format shown in Figure 3–3.

Figure 3–3: Branch Instruction Format

A Branch format instruction contains a 6-bit opcode field, one 5-bit register address field (Ra), and a 21-bit signed displacement field.

The displacement is treated as a longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits and added to the updated PC to form the target virtual address. Overflow is ignored in this calculation. The target virtual address (va) is computed as follows:

\[
va \leftarrow PC + \{4 \times \text{SEXT}(\text{Branch\_disp})\}
\]

3.3.3 Operate Instruction Format

The Operate format is used for instructions that perform integer register to integer register operations. The Operate format allows the specification of one destination operand and two source operands. One of the source operands can be a literal constant. The Operate format in Figure 3–4 shows the two cases when bit <12> of the instruction is 0 and 1.

Figure 3–4: Operate Instruction Format

An Operate format instruction contains a 6-bit opcode field and a 7-bit function field. Unused function codes for those opcodes defined as reserved in the Version 5 Alpha architecture specification (May 1992) produce an illegal instruction trap. Those opcodes are 01, 02, 03, 04, 05, 06, 07, 0A, 0C, 0D, 0E, 14, 19, 1B, 1D, 1E, and 1F. For other opcodes, unused function codes produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.
There are three operand fields, Ra, Rb, and Rc.

The Ra field specifies a source operand. Symbolically, the integer Rav operand is formed as follows:

\[
\text{IF } \text{inst}<25:21> \text{ EQ 31 THEN} \\
\text{Rav } \leftarrow \ 0 \\
\text{ELSE} \\
\text{Rav } \leftarrow \ Ra \\
\text{END}
\]

The Rb field specifies a source operand. Integer operands can specify a literal or an integer register using bit <12> of the instruction.

If bit <12> of the instruction is 0, the Rb field specifies a source register operand.

If bit <12> of the instruction is 1, an 8-bit zero-extended literal constant is formed by bits <20:13> of the instruction. The literal is interpreted as a positive integer between 0 and 255 and is zero-extended to 64 bits. Symbolically, the integer Rbv operand is formed as follows:

\[
\text{IF } \text{inst}<12> \text{ EQ 1 THEN} \\
\text{Rbv } \leftarrow \ \text{ZEXT}(\text{inst}<20:13>) \\
\text{ELSE} \\
\text{IF } \text{inst}<20:16> \text{ EQ 31 THEN} \\
\text{Rbv } \leftarrow \ 0 \\
\text{ELSE} \\
\text{Rbv } \leftarrow \ Rb \\
\text{END} \\
\text{END}
\]

The Rc field specifies a destination operand.

### 3.3.4 Floating-Point Operate Instruction Format

The Floating-point Operate format is used for instructions that perform floating-point register to floating-point register operations. The Floating-point Operate format allows the specification of one destination operand and two source operands. The Floating-point Operate format is shown in Figure 3–5.

**Figure 3–5: Floating-Point Operate Instruction Format**

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Fa</td>
<td>Fb</td>
<td>Function</td>
<td>Fc</td>
<td></td>
</tr>
</tbody>
</table>

A Floating-point Operate format instruction contains a 6-bit opcode field and an 11-bit function field. Unused function codes for those opcodes defined as reserved in the Version 5 Alpha architecture specification (May 1992) produce an illegal instruction trap. Those opcodes are 01, 02, 03, 04, 05, 06, 07, 14, 19, 1B, 1D, 1E, and 1F. For other opcodes, unused function codes produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.
There are three operand fields, Fa, Fb, and Fc. Each operand field specifies either an integer or floating-point operand as defined by the instruction.

The Fa field specifies a source operand. Symbolically, the Fav operand is formed as follows:

\[
\begin{align*}
\text{IF } \text{inst}[25:21] &\text{ EQ 31 THEN} \\
\text{Fav} &\leftarrow 0 \\
\text{ELSE} \\
\text{Fav} &\leftarrow \text{Fa} \\
\text{END}
\end{align*}
\]

The Fb field specifies a source operand. Symbolically, the Fbv operand is formed as follows:

\[
\begin{align*}
\text{IF } \text{inst}[20:16] &\text{ EQ 31 THEN} \\
\text{Fbv} &\leftarrow 0 \\
\text{ELSE} \\
\text{Fbv} &\leftarrow \text{Fb} \\
\text{END}
\end{align*}
\]

Note

Neither Fa nor Fb can be a literal in Floating-point Operate instructions.

The Fc field specifies a destination operand.

### 3.3.4.1 Floating-Point Convert Instructions

Floating-point Convert instructions use a subset of the Floating-point Operate format and perform register-to-register conversion operations. The Fb operand specifies the source; the Fa field must be F31.

### 3.3.4.2 Floating-Point/Integer Register Moves

Instructions that move data between a floating-point register file and an integer register file are a subset of the Floating-point Operate format. The unused source field must be 31.

### 3.3.5 PALcode Instruction Format

The Privileged Architecture Library (PALcode) format is used to specify extended processor functions. It has the format shown in Figure 3–6.

**Figure 3–6: PALcode Instruction Format**

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>PALcode Function</td>
<td></td>
</tr>
</tbody>
</table>

The 26-bit PALcode function field specifies the operation.

The source and destination operands for PALcode instructions are supplied in fixed registers that are specified in the individual instruction descriptions.
An opcode of zero and a PALcode function of zero specify the HALT instruction.
Chapter 4

Instruction Descriptions (I)

4.1 Instruction Set Overview

This chapter describes the instructions implemented by the Alpha architecture. The instruction set is divided into the following sections:

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer load and store</td>
<td>4.2</td>
</tr>
<tr>
<td>Integer control</td>
<td>4.3</td>
</tr>
<tr>
<td>Integer arithmetic</td>
<td>4.4</td>
</tr>
<tr>
<td>Logical and shift</td>
<td>4.5</td>
</tr>
<tr>
<td>Byte manipulation</td>
<td>4.6</td>
</tr>
<tr>
<td>Floating-point load and store</td>
<td>4.8</td>
</tr>
<tr>
<td>Floating-point control</td>
<td>4.9</td>
</tr>
<tr>
<td>Floating-point operate</td>
<td>4.10</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>4.11</td>
</tr>
<tr>
<td>Multimedia (graphics and video)</td>
<td>4.13</td>
</tr>
</tbody>
</table>

Within each major section, closely related instructions are combined into groups and described together. The instruction group description is composed of the following:

- The group name
- The format of each instruction in the group, which includes the name, access type, and data type of each instruction operand
- The operation of the instruction
- Exceptions specific to the instruction
- The instruction mnemonic and name of each instruction in the group
- Qualifiers specific to the instructions in the group
- A description of the instruction operation
- Optional programming examples and optional notes on the instruction
4.1.1 Subsetting Rules

An instruction that is omitted in a subset implementation of the Alpha architecture is not performed in either hardware or PAL code. System software may provide emulation routines for subsetted instructions.

4.1.1.1 Floating-Point Subsets

Floating-point support is optional on an Alpha processor. An implementation that supports floating-point must implement the following:

- The 32 floating-point registers
- The Floating-point Control Register (FPCR) and the instructions to access it.
- The floating-point branch instructions
- The floating-point copy sign (CPYSx) instructions.
- The floating-point convert instructions.
- The floating-point conditional move instruction (FCMOV).
- The integer register to floating-point register, and floating-point register to integer register move instructions.
- The S_floating and T_floating memory operations.

Software Note:

A system that will not support floating-point operations is still required to provide the 32 floating-point registers, the Floating-point Control Register (FPCR) and the instructions to access it, and the T_floating memory operations if the system intends to support the OpenVMS Alpha operating system. This requirement facilitates the implementation of a floating-point emulator and simplifies context-switching.

In addition, floating-point support requires at least one of the following subset groups:

1. VAX Floating-point Operate and Memory instructions (F_ and G_floating).
2. IEEE Floating-point Operate instructions (S_ and T_floating). Within this group, an implementation can choose to include or omit separately the ability to perform IEEE rounding to plus infinity and minus infinity.

Note: if one instruction in a group is provided, all other instructions in that group must be provided. An implementation with full floating-point support includes both groups; a subset floating-point implementation supports only one of these groups. The individual instruction descriptions indicate whether an instruction can be subsetted.
4.1.2 Software Emulation Rules

General-purpose layered and application software that executes in User mode may assume that certain loads (LDL, LDQ, LDF, LDG, LDS, and LDT) and certain stores (STL, STQ, STF, STG, STL, and STT) of unaligned data are emulated by system software. General-purpose layered and application software that executes in User mode may assume that subsetted instructions are emulated by system software. Frequent use of emulation may be significantly slower than using alternative code sequences.

Emulation of loads and stores of unaligned data and subsetted instructions need not be provided in privileged access modes. System software that supports special-purpose dedicated applications need not provide emulation in User mode if emulation is not needed for correct execution of the special-purpose applications.

4.1.3 Opcode Qualifiers

Some Operate format and Floating-point Operate format instructions have several variants. For example, for the VAX formats, Add F floating (ADDF) is supported with and without floating underflow enabled, and with either chopped or VAX rounding. For IEEE formats, IEEE unbiased rounding, chopped, round toward plus infinity, and round toward minus infinity can be selected.

The different variants of such instructions are denoted by opcode qualifiers, which consist of a slash (/) followed by a string of selected qualifiers. Each qualifier is denoted by a single character as shown in Table 4–1. The opcodes for each qualifier are listed in Appendix C.

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Chopped rounding</td>
</tr>
<tr>
<td>D</td>
<td>Rounding mode dynamic</td>
</tr>
<tr>
<td>M</td>
<td>Round toward minus infinity</td>
</tr>
<tr>
<td>I</td>
<td>Inexact result enable</td>
</tr>
<tr>
<td>S</td>
<td>Software completion enable</td>
</tr>
<tr>
<td>U</td>
<td>Floating underflow enable</td>
</tr>
<tr>
<td>V</td>
<td>Integer overflow enable</td>
</tr>
</tbody>
</table>

The default values are normal rounding, software completion disabled, inexact result disabled, floating underflow disabled, and integer overflow disabled.
4.2 Memory Integer Load/Store Instructions

The instructions in this section move data between the integer registers and memory. They use the Memory instruction format. The instructions are summarized in Table 4–2.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>Load Address</td>
</tr>
<tr>
<td>LDAH</td>
<td>Load Address High</td>
</tr>
<tr>
<td>LDBU</td>
<td>Load Zero-Extended Byte from Memory to Register</td>
</tr>
<tr>
<td>LDL</td>
<td>Load Sign-Extended Longword</td>
</tr>
<tr>
<td>LDL_L</td>
<td>Load Sign-Extended Longword Locked</td>
</tr>
<tr>
<td>LDQ</td>
<td>Load Quadword</td>
</tr>
<tr>
<td>LDQ_L</td>
<td>Load Quadword Locked</td>
</tr>
<tr>
<td>LDQ_U</td>
<td>Load Quadword Unaligned</td>
</tr>
<tr>
<td>LDWU</td>
<td>Load Zero-Extended Word from Memory to Register</td>
</tr>
<tr>
<td>STB</td>
<td>Store Byte</td>
</tr>
<tr>
<td>STL</td>
<td>Store Longword</td>
</tr>
<tr>
<td>STL_C</td>
<td>Store Longword Conditional</td>
</tr>
<tr>
<td>STQ</td>
<td>Store Quadword</td>
</tr>
<tr>
<td>STQ_C</td>
<td>Store Quadword Conditional</td>
</tr>
<tr>
<td>STQ_U</td>
<td>Store Quadword Unaligned</td>
</tr>
<tr>
<td>STW</td>
<td>Store Word</td>
</tr>
</tbody>
</table>
4.2.1 Load Address

Format:

LDAx Ra.wq,disp.ab(Rb.ab) !Memory format

Operation:

Ra ← Rbv + SEXT(disp) !LDA
Ra ← Rbv + SEXT(disp*65536) !LDAH

Exceptions:

None

Instruction mnemonics:

LDA Load Address
LDAH Load Address High

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement for LDA, and 65536 times the sign-extended 16-bit displacement for LDAH. The 64-bit result is written to register Ra.
4.2.2 Load Memory Data into Integer Register

Format:

LDx Ra.wq,disp.ab(Rb.ab) !Memory format

Operation:

va ← \{Rbv + SEXT(disp)\}

CASE

big_endian_data: va’ ← va XOR 0002 !LDQ
big_endian_data: va’ ← va XOR 1002 !LDL
big_endian_data: va’ ← va XOR 1102 !LDWU
big_endian_data: va’ ← va XOR 1112 !LDBU

little_endian_data: va’ ← va

ENDCASE

Ra ← (va’)<63:0> !LDQ
Ra ← SEXT((va’)<31:0>) !LDL
Ra ← ZEXT((va’)<15:0>) !LDWU
Ra ← ZEXT((va’)<07:0>) !LDBU

Exceptions:

Access Violation
Alignment
Fault on Read
Translation Not Valid

Instruction mnemonics:

LDBU Load Zero-Extended Byte from Memory to Register
LDL Load Sign-Extended Longword from Memory to Register
LDQ Load Quadword from Memory to Register
LDWU Load Zero-Extended Word from Memory to Register

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. For a big-endian access, the indicated bits are inverted, and any memory management fault is reported for va (not va’).
In the case of LDQ and LDL, the source operand is fetched from memory, sign-extended, and written to register Ra.

In the case of LDWU and LDBU, the source operand is fetched from memory, zero-extended, and written to register Ra.

In all cases, if the data is not naturally aligned, an alignment exception is generated.

Notes:

• The word or byte that the LDWU or LDBU instruction fetches from memory is placed in the low (rightmost) word or byte of Ra, with the remaining 6 or 7 bytes set to zero.

• Accesses have byte granularity.

• For big-endian access with LDWU or LDBU, the word/byte remains in the rightmost part of Ra, but the va sent to memory has the indicated bits inverted. See Operation section, above.

• No sparse address space mechanisms are allowed with the LDWU and LDBU instructions.

Implementation Notes:

• Depending on an address space region’s caching policy, implementations may read a (partial) cache block in order to do word/byte stores. This may only be done in regions that have memory-like behavior.

• Implementations are expected to provide sufficient low-order address bits and length-of-access information to devices on I/O buses. But, strictly speaking, this is outside the scope of architecture.
4.2.3 Load Unaligned Memory Data into Integer Register

Format:

LDQ_U Ra.wq,disp.ab(Rb.ab) !Memory format

Operation:

va ← \{Rbv + SEXT(disp)} AND NOT 7\}
Ra ← (va)<63:0>

Exceptions:

Access Violation
Fault on Read
Translation Not Valid

Instruction mnemonics:

LDQ_U Load Unaligned Quadword from Memory to Register

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement, then the low-order three bits are cleared. The source operand is fetched from memory and written to register Ra.
4.2.4 Load Memory Data into Integer Register Locked

Format:

```
LDx_L      Ra.wq,disp.ab(Rb.ab)           !Memory format
```

Operation:

```
va ← {Rbv + SEXT(disp)}
CASE
    big_endian_data: va' ← va XOR 000 2     ! LDQ_L
    big_endian_data: va' ← va XOR 100 2     ! LDL_L
    little_endian_data: va' ← va           ! LDL_L
ENDCASE
lock_flag ← 1
locked_physical_address ← PHYSICAL_ADDRESS(va)
Ra ← SEXT((va')<31:0>)                   ! LDL_L
Ra ← (va)<63:0>                         ! LDQ_L
```

Exceptions:

- Access Violation
- Alignment
- Fault on Read
- Translation Not Valid

Instruction mnemonics:

- LDL_L Load Sign-Extended Longword from Memory to Register Locked
- LDQ_L Load Quadword from Memory to Register Locked

Qualifiers:

- None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. For a big-endian longword access, va<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va'). The source operand is fetched from memory, sign-extended for LDL_L, and written to register Ra.
When a LDx_L instruction is executed without faulting, the processor records the target physical address in a per-processor locked_physical_address register and sets the per-processor lock_flag.

If the per-processor lock_flag is (still) set when a STx_C instruction is executed (accessing within the same 16-byte naturally aligned block as the LDx_L), the store occurs; otherwise, it does not occur, as described for the STx_C instructions. The behavior of a STx_C instruction is UNPREDICTABLE, as described in Section 4.2.5, when it does not access the same 16-byte naturally aligned block as the LDx_L.

Processor A causes the clearing of a set lock_flag in processor B by doing any of the following in B’s locked range of physical addresses: a successful store, a successful store_conditional, or executing a WH64 instruction that modifies data on processor B. A processor’s locked range is the aligned block of 2**N bytes that includes the locked_physical_address. The 2**N value is implementation dependent. It is at least 16 (minimum lock range is an aligned 16-byte block) and is at most the page size for that implementation (maximum lock range is one physical page).

A processor’s lock_flag is also cleared if that processor encounters a CALL_PAL REI, CALL_PAL rti, or CALL_PAL rfe instruction. It is UNPREDICTABLE whether or not a processor’s lock_flag is cleared on any other CALL_PAL instruction. It is UNPREDICTABLE whether a processor’s lock_flag is cleared by that processor executing a normal load or store instruction. It is UNPREDICTABLE whether a processor’s lock_flag is cleared by that processor executing a taken branch (including BR, BSR, and jumps); conditional branches that fall through do not clear the lock_flag. It is UNPREDICTABLE whether a processor’s lock_flag is cleared by that processor executing a WH64 or ECB instruction.

The sequence:

LDx_L
Modify
STx_C
BEQ xxx

when executed on a given processor, does an atomic read-modify-write of a datum in shared memory if the branch falls through. If the branch is taken, the store did not modify memory and the sequence may be repeated until it succeeds.

Notes:

• LDx_L instructions do not check for write access; hence a matching STx_C may take an access-violation or fault-on-write exception.

Executing a LDx_L instruction on one processor does not affect any architecturally visible state on another processor, and in particular cannot cause a STx_C on another processor to fail.

LDx_L and STx_C instructions need not be paired. In particular, an LDx_L may be followed by a conditional branch: on the fall-through path an STx_C is executed, whereas on the taken path no matching STx_C is executed.
If two LDx_L instructions execute with no intervening STx_C, the second one overwrites the state of the first one. If two STx_C instructions execute with no intervening LDx_L, the second one always fails because the first clears lock_flag.

- Software will not emulate unaligned LDx_L instructions.
- If the virtual and physical addresses for a LDx_L and STx_C sequence are not within the same naturally aligned 16-byte sections of virtual and physical memory, that sequence may always fail, or may succeed despite another processor’s store to the lock range; hence, no useful program should do this.
- If any other memory access (ECB, LDx, LDQ_U, STx, STQ_U, WH64) is executed on the given processor between the LDx_L and the STx_C, the sequence above may always fail on some implementations; hence, no useful program should do this.
- If a branch is taken between the LDx_L and the STx_C, the sequence above may always fail on some implementations; hence, no useful program should do this. (CMOVxx may be used to avoid branching.)
- If a subsetted instruction (for example, floating-point) is executed between the LDx_L and the STx_C, the sequence above may always fail on some implementations, because of the Illegal Instruction Trap; hence, no useful program should do this.
- If an instruction with an unused function code is executed between the LDx_L and the STx_C, the sequence above may always fail on some implementations because an instruction with an unused function code is UNPREDICTABLE.
- If a large number of instructions are executed between the LDx_L and the STx_C, the sequence above may always fail on some implementations because of a timer interrupt always clearing the lock_flag before the sequence completes; hence, no useful program should do this.
- Hardware implementations are encouraged to lock no more than 128 bytes. Software implementations are encouraged to separate locked locations by at least 128 bytes from other locations that could potentially be written by another processor while the first location is locked.
- Execution of a WH64 instruction on processor A to a region within the lock range of processor B, where the execution of the WH64 changes the contents of memory, causes the lock_flag on processor B to be cleared. If the WH64 does not change the contents of memory on processor B, it need not clear the lock_flag.

Implementation Notes:

Implementations that impede the mobility of a cache block on LDx_L, such as that which may occur in a Read for Ownership cache coherency protocol, may release the cache block and make the subsequent STx_C fail if a branch-taken or memory instruction is executed on that processor.

All implementations should guarantee that at least 40 non-subsetted operate instructions can be executed between timer interrupts.
4.2.5 Store Integer Register Data into Memory Conditional

Format:

\[ \text{STx}_C \quad \text{Ra.mx,disp.ab(Rb.ab)} \quad \text{!Memory format} \]

Operation:

\[ \text{va} \leftarrow \{Rbv + \text{SEXT(disp)}\} \]

CASE

\[ \begin{align*}
\text{big_endian_data: } & \quad \text{va}' \leftarrow \text{va XOR 000}_2 \quad \text{! STQ}_C \\
\text{big_endian_data: } & \quad \text{va}' \leftarrow \text{va XOR 100}_2 \quad \text{! STL}_C \\
\text{little_endian_data: } & \quad \text{va}' \leftarrow \text{va} \quad \text{! STL}_C
\end{align*} \]

ENDCASE

IF lock_flag EQ 1 THEN

\[ \begin{align*}
(\text{va}')<31:0> & \leftarrow \text{Rav}<31:0> \quad \text{! STL}_C \\
(\text{va}) & \leftarrow \text{Rav} \quad \text{! STQ}_C \\
\text{Ra} & \leftarrow \text{lock_flag} \\
\text{lock_flag} & \leftarrow 0
\end{align*} \]

Exceptions:

- Access Violation
- Fault on Write
- Alignment
- Translation Not Valid

Instruction mnemonics:

- \text{STL}_C \quad \text{Store Longword from Register to Memory Conditional}
- \text{STQ}_C \quad \text{Store Quadword from Register to Memory Conditional}

Qualifiers:

- None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. For a big-endian longword access, va<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va').

If the lock_flag is set and the address meets the following constraints relative to the address specified by the preceding LDx_L instruction, the Ra operand is written...
to memory at this address. If the address meets the following constraints but the 
lock_flag is not set, a zero is returned in Ra and no write to memory occurs. The 
constraints are:

- The computed virtual address must specify a location within the naturally 
  aligned 16-byte block in virtual memory accessed by the preceding LDx_L 
  instruction.

- The resultant physical address must specify a location within the naturally 
  aligned 16-byte block in physical memory accessed by the preceding LDx_L 
  instruction.

If those addressing constraints are not met, it is UNPREDICTABLE whether the 
STx_C instruction succeeds or fails, regardless of the state of the lock_flag, unless 
the lock_flag is cleared as described in the next paragraph.

Whether or not the addressing constraints are met, a zero is returned and no write 
to memory occurs if the lock_flag was cleared by execution on a processor of a CALL_ 
PAL REI, CALL_PAL rti, CALL_PAL rfe, or STx_C, after the most recent execution 
on that processor of a LDx_L instruction (in processor issue sequence).

In all cases, the lock_flag is set to zero at the end of the operation.

Notes:

- Software will not emulate unaligned STx_C instructions.

- Each implementation must do the test and store atomically, as illustrated in the 
  following two examples. (See Section 5.6.1 for complete information.)

  - If two processors attempt STx_C instructions to the same lock range and that 
    lock range was accessed by both processors’ preceding LDx_L instructions, 
    exactly one of the stores succeeds.

  - A processor executes a LDx_L/STx_C sequence and includes an MB between 
    the LDx_L to a particular address and the successful STx_C to a different 
    address (one that meets the constraints required for predictable behavior). 
    That instruction sequence establishes an access order under which a store 
    operation by another processor to that lock range occurs before the LDx_L or 
    after the STx_C.

- If the virtual and physical addresses for a LDx_L and STx_C sequence are 
  not within the same naturally aligned 16-byte sections of virtual and physical 
  memory, that sequence may always fail, or may succeed despite another 
  processor’s store to the lock range; hence, no useful program should do this.

- The following sequence should not be used:

Instruction Descriptions (I) 4–13
try_again: LDQ_L R1,x
<modify R1>
STQ_C R1,x
BEQ R1, try_again

That sequence penalizes performance when the STQ_C succeeds, because the sequence contains a backward branch, which is predicted to be taken in the Alpha architecture. In the case where the STQ_C succeeds and the branch will actually fall through, that sequence incurs unnecessary delay due to a mispredicted backward branch. Instead, a forward branch should be used to handle the failure case, as shown in Section 5.5.2.

Software Note:

If the address specified by a STx_C instruction does not match the one given in the preceding LDx_L instruction, an MB is required to guarantee ordering between the two instructions.

Hardware/Software Implementation Note:

STQ_C is used in the first Alpha implementations to access the MailBox Pointer Register (MBPR). In this special case, the effect of the STQ_C is well defined (that is, not UNPREDICTABLE) even though the preceding LDx_L did not specify the address of the MBPR. The effect of STx_C in this special case may vary from implementation to implementation.

Implementation Notes:

A STx_C must propagate to the point of coherency, where it is guaranteed to prevent any other store from changing the state of the lock bit, before its outcome can be determined.

If an implementation could encounter a TB or cache miss on the data reference of the STx_C in the sequence above (as might occur in some shared I- and D-stream direct-mapped TBs/caches), it must be able to resolve the miss and complete the store without always failing.
4.2.6 Store Integer Register Data into Memory

Format:

\[ \text{ST}x \quad \text{Ra}.rx,\text{disp.ab(Rb.ab)} \quad \text{!Memory format} \]

Operation:

\[ \text{va} \leftarrow \{ \text{Rbv} + \text{SEXT(disp)} \} \]

CASE

\[ \begin{align*}
\text{big\_endian\_data:} & \quad \text{va}' \leftarrow \text{va} \oplus 000_2 \quad \text{!STQ} \\
\text{big\_endian\_data:} & \quad \text{va}' \leftarrow \text{va} \oplus 100_2 \quad \text{!STL} \\
\text{big\_endian\_data:} & \quad \text{va}' \leftarrow \text{va} \oplus 110_2 \quad \text{!STW} \\
\text{big\_endian\_data:} & \quad \text{va}' \leftarrow \text{va} \oplus 111_2 \quad \text{!STB} \\
\text{little\_endian\_data:} & \quad \text{va}' \leftarrow \text{va} \\
\end{align*} \]

ENDCASE

\[ \begin{align*}
\text{(va')} & \leftarrow \text{Rav} \quad \text{!STQ} \\
\text{(va')}<31:00> & \leftarrow \text{Rav}<31:0> \quad \text{!STL} \\
\text{(va')}<15:00> & \leftarrow \text{Rav}<15:0> \quad \text{!STW} \\
\text{(va')}<07:00> & \leftarrow \text{Rav}<07:0> \quad \text{!STB} \\
\end{align*} \]

Exceptions:

Access Violation
Alignment
Fault on Read
Translation Not Valid

Instruction mnemonics:

- **STB**: Store Byte from Register to Memory
- **STL**: Store Longword from Register to Memory
- **STQ**: Store Quadword from Register to Memory
- **STW**: Store Word from Register to Memory

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. For a big-endian access, the indicated bits are inverted, and any memory management fault is reported for va (not va').
The Ra operand is written to memory at this address. If the data is not naturally aligned, an alignment exception is generated.

Notes:

• The word or byte that the STB or STW instruction stores to memory comes from the low (rightmost) byte or word of Ra.
• Accesses have byte granularity.
• For big-endian access with STB or STW, the byte/word remains in the rightmost part of Ra, but the va sent to memory has the indicated bits inverted. See Operation section, above.
• No sparse address space mechanisms are allowed with the STB and STW instructions.

Implementation Notes:

• Depending on an address space region’s caching policy, implementations may read a (partial) cache block in order to do byte/word stores. This may only be done in regions that have memory-like behavior.
• Implementations are expected to provide sufficient low-order address bits and length-of-access information to devices on I/O buses. But, strictly speaking, this is outside the scope of architecture.
4.2.7 Store Unaligned Integer Register Data into Memory

Format:

```
STQ_U   Ra.rq,disp.ab(Rb.ab) !Memory format
```

Operation:

```
va ← {{Rbv + SEXT(disp)} AND NOT 7}
(va)<63:0> ← Rav<63:0>
```

Exceptions:

- Access Violation
- Fault on Write
- Translation Not Valid

Instruction mnemonics:

```
STQ_U   Store Unaligned Quadword from Register to Memory
```

Qualifiers:

None

Description:

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement, then clearing the low order three bits. The Ra operand is written to memory at this address.
4.3 Control Instructions

Alpha provides integer conditional branch, unconditional branch, branch to subroutine, and jump instructions. The PC used in these instructions is the updated PC, as described in Section 3.1.1.

To allow implementations to achieve high performance, the Alpha architecture includes explicit hints based on a branch-prediction model:

1. For many implementations of computed branches (JSR/RET/JMP), there is a substantial performance gain in forming a good guess of the expected target I-cache address before register Rb is accessed.
2. For many implementations, the first-level (or only) I-cache is no bigger than a page (8 KB to 64 KB).
3. Correctly predicting subroutine returns is important for good performance. Some implementations will therefore keep a small stack of predicted subroutine return I-cache addresses.

The Alpha architecture provides three kinds of branch-prediction hints: likely target address, return-address stack action, and conditional branch-taken.

For computed branches, the otherwise unused displacement field contains a function code (JMP/JSR/RET/JSR_COROUTINE), and, for JSR and JMP, a field that statically specifies the 16 low bits of the most likely target address. The PC-relative calculation using these bits can be exactly the PC-relative calculation used in unconditional branches. The low 16 bits are enough to specify an I-cache block within the largest possible Alpha page and hence are expected to be enough for branch-prediction logic to start an early I-cache access for the most likely target.

For all branches, hint or opcode bits are used to distinguish simple branches, subroutine calls, subroutine returns, and coroutine links. These distinctions allow branch-predict logic to maintain an accurate stack of predicted return addresses.

For conditional branches, the sign of the target displacement is used as a taken/fall-through hint. The instructions are summarized in Table 4–3.
### Table 4–3: Control Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ</td>
<td>Branch if Register Equal to Zero</td>
</tr>
<tr>
<td>BGE</td>
<td>Branch if Register Greater Than or Equal to Zero</td>
</tr>
<tr>
<td>BGT</td>
<td>Branch if Register Greater Than Zero</td>
</tr>
<tr>
<td>BLBC</td>
<td>Branch if Register Low Bit Is Clear</td>
</tr>
<tr>
<td>BLBS</td>
<td>Branch if Register Low Bit Is Set</td>
</tr>
<tr>
<td>BLE</td>
<td>Branch if Register Less Than or Equal to Zero</td>
</tr>
<tr>
<td>BLT</td>
<td>Branch if Register Less Than Zero</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch if Register Not Equal to Zero</td>
</tr>
<tr>
<td>BR</td>
<td>Unconditional Branch</td>
</tr>
<tr>
<td>BSR</td>
<td>Branch to Subroutine</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to Subroutine</td>
</tr>
<tr>
<td>RET</td>
<td>Return from Subroutine</td>
</tr>
<tr>
<td>JSR_COROUTINE</td>
<td>Jump to Subroutine Return</td>
</tr>
</tbody>
</table>
4.3.1 Conditional Branch

Format:

Bxx Ra.rq,disp.al !Branch format

Operation:

(update PC)
va ← PC + {4*SEXT(disp)}
IF TEST(Rav, Condition_based_on_Opcode) THEN
   PC ← va

Exceptions:

None

Instruction mnemonics:

BEQ    Branch if Register Equal to Zero
BGE    Branch if Register Greater Than or Equal to Zero
BGT    Branch if Register Greater Than Zero
BLBC   Branch if Register Low Bit Is Clear
BLBS   Branch if Register Low Bit Is Set
BLE    Branch if Register Less Than or Equal to Zero
BLT    Branch if Register Less Than Zero
BNE    Branch if Register Not Equal to Zero

Qualifiers:

None

Description:

Register Ra is tested. If the specified relationship is true, the PC is loaded with the target virtual address; otherwise, execution continues with the next sequential instruction.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address.

The conditional branch instructions are PC-relative only. The 21-bit signed displacement gives a forward/backward branch distance of +/- 1M instructions.
The test is on the signed quadword integer interpretation of the register contents; all 64 bits are tested.

Notes:

- Forward conditional branches (positive displacement) are predicted to fall through. Backward conditional branches (negative displacement) are predicted to be taken. Conditional branches do not affect a predicted return address stack.
4.3.2 Unconditional Branch

Format:

\[ \text{BxR Ra.wq,disp.al} \]

!Branch format

Operation:

\[
\begin{align*}
\{\text{update PC}\} \\
\text{Ra} & \leftarrow \text{PC} \\
\text{PC} & \leftarrow \text{PC} + \{4\text{SEXT(disp)}\}
\end{align*}
\]

Exceptions:

None

Instruction mnemonics:

BR Unconditional Branch
BSR Branch to Subroutine

Qualifiers:

None

Description:

The PC of the following instruction (the updated PC) is written to register Ra, and then the PC is loaded with the target address.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address.

The unconditional branch instructions are PC-relative. The 21-bit signed displacement gives a forward/backward branch distance of +/- 1M instructions.

PC-relative addressability can be established by:

\[ \text{BR Rx,L1} \]

L1:

Notes:

- BR and BSR do identical operations. They only differ in hints to possible branch-prediction logic. BSR is predicted as a subroutine call (pushes the return address on a branch-prediction stack), whereas BR is predicted as a branch (no push).
4.3.3 Jumps

Format:

\[
\text{mnemonic Ra},wq,(Rb),ab,\text{hint} !\text{Memory format}
\]

Operation:

\{
\begin{align*}
\text{update PC} \\
va & \leftarrow \text{Rbv AND \{NOT 3\}} \\
\text{Ra} & \leftarrow \text{PC} \\
\text{PC} & \leftarrow va
\end{align*}
\}

Exceptions:

None

Instruction mnemonics:

\begin{align*}
\text{JMP} & \quad \text{Jump} \\
\text{JSR} & \quad \text{Jump to Subroutine} \\
\text{RET} & \quad \text{Return from Subroutine} \\
\text{JSR\_COROUTINE} & \quad \text{Jump to Subroutine Return}
\end{align*}

Qualifiers:

None

Description:

The PC of the instruction following the Jump instruction (the updated PC) is written to register Ra, and then the PC is loaded with the target virtual address.

The new PC is supplied from register Rb. The low two bits of Rb are ignored. Ra and Rb may specify the same register; the target calculation using the old value is done before the new value is assigned.

All Jump instructions do identical operations. They only differ in hints to possible branch-prediction logic. The displacement field of the instruction is used to pass this information. The four different “opcodes” set different bit patterns in disp<15:14>, and the hint operand sets disp<13:0>.
These bits are intended to be used as shown in Table 4–4.

**Table 4–4: Jump Instructions Branch Prediction**

<table>
<thead>
<tr>
<th>disp&lt;15:14&gt;</th>
<th>Meaning</th>
<th>Predicted Target&lt;15:0&gt;</th>
<th>Prediction Stack Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>JMP</td>
<td>PC + 4*disp&lt;13:0&gt;</td>
<td>–</td>
</tr>
<tr>
<td>01</td>
<td>JSR</td>
<td>PC + 4*disp&lt;13:0&gt;</td>
<td>Push PC</td>
</tr>
<tr>
<td>10</td>
<td>RET</td>
<td>Prediction stack</td>
<td>Pop</td>
</tr>
<tr>
<td>11</td>
<td>JSR_COROUTINE</td>
<td>Prediction stack</td>
<td>Pop, push PC</td>
</tr>
</tbody>
</table>

The design in Table 4–4 allows specification of the low 16 bits of a likely longword target address (enough bits to start a useful I-cache access early), and also allows distinguishing call from return (and from the other two less frequent operations).

Note that the above information is used only as a hint; correct setting of these bits can improve performance but is not needed for correct operation. See Appendix A for more information on branch prediction.

An unconditional long jump can be performed by:

```
JMP R31, (Rb), hint
```

Coroutine linkage can be performed by specifying the same register in both the Ra and Rb operands. When disp<15:14> equals ‘10’ (RET) or ‘11’ (JSR_COROUTINE) (that is, the target address prediction, if any, would come from a predictor implementation stack), then bits <13:0> are reserved for software and must be ignored by all implementations. All encodings for bits <13:0> are used by Digital software or Reserved to Digital, as follows:

**Encoding Meaning**

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000&lt;sub&gt;16&lt;/sub&gt;</td>
<td>Indicates non-procedure return</td>
</tr>
<tr>
<td>0001&lt;sub&gt;16&lt;/sub&gt;</td>
<td>Indicates procedure return</td>
</tr>
</tbody>
</table>

All other encodings are reserved to Digital.
4.4 Integer Arithmetic Instructions

The integer arithmetic instructions perform add, subtract, multiply, signed and unsigned compare, and bit count operations.

The integer instructions are summarized in Table 4–5.

Table 4–5: Integer Arithmetic Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add Quadword/Longword</td>
</tr>
<tr>
<td>S4ADD</td>
<td>Scaled Add by 4</td>
</tr>
<tr>
<td>S8ADD</td>
<td>Scaled Add by 8</td>
</tr>
<tr>
<td>CMPEQ</td>
<td>Compare Signed Quadword Equal</td>
</tr>
<tr>
<td>CMPLT</td>
<td>Compare Signed Quadword Less Than</td>
</tr>
<tr>
<td>CMPLE</td>
<td>Compare Signed Quadword Less Than or Equal</td>
</tr>
<tr>
<td>CTLZ</td>
<td>Count leading zero</td>
</tr>
<tr>
<td>CTPOP</td>
<td>Count population</td>
</tr>
<tr>
<td>CTTZ</td>
<td>Count trailing zero</td>
</tr>
<tr>
<td>CMPULT</td>
<td>Compare Unsigned Quadword Less Than</td>
</tr>
<tr>
<td>CMPULE</td>
<td>Compare Unsigned Quadword Less Than or Equal</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply Quadword/Longword</td>
</tr>
<tr>
<td>UMULH</td>
<td>Multiply Quadword Unsigned High</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract Quadword/Longword</td>
</tr>
<tr>
<td>S4SUB</td>
<td>Scaled Subtract by 4</td>
</tr>
<tr>
<td>S8SUB</td>
<td>Scaled Subtract by 8</td>
</tr>
</tbody>
</table>

There is no integer divide instruction. Division by a constant can be done via UMULH; division by a variable can be done via a subroutine. See Appendix A.
### 4.4.1 Longword Add

**Format:**

```
ADDL Ra.rl,Rb.rl,Rc.wq       !Operate format
ADDL Ra.rl,#b.ib,Rc.wq       !Operate format
```

**Operation:**

```
Rc ← SEXT( (Rav + Rbv)<31:0>)
```

**Exceptions:**

- Integer Overflow

**Instruction mnemonics:**

ADDL Add Longword

**Qualifiers:**

- Integer Overflow Enable (/V)

**Description:**

Register Ra is added to register Rb or a literal, and the sign-extended 32-bit sum is written to Rc.

The high order 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit sum. Overflow detection is based on the longword sum Rav<31:0> + Rbv<31:0>.
4.4.2 Scaled Longword Add

Format:

SxADDL Ra.rl,Rb.rq,Rc.wq !Operate format
SxADDL Ra.rl,#b.ib,Rc.wq !Operate format

Operation:

CASE
  S4ADDL: Rc ← SEXT (((LEFT_SHIFT(Rav,2)) + Rbv)<31:0>)
  S8ADDL: Rc ← SEXT (((LEFT_SHIFT(Rav,3)) + Rbv)<31:0>)
ENDCASE

Exceptions:

None

Instruction mnemonics:

S4ADDL Scaled Add Longword by 4
S8ADDL Scaled Add Longword by 8

Qualifiers:

None

Description:

Register Ra is scaled by 4 (for S4ADDL) or 8 (for S8ADDL) and is added to register
Rb or a literal, and the sign-extended 32-bit sum is written to Rc.

The high 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the
truncated 32-bit sum.
4.4.3 Quadword Add

Format:

\[
\text{ADDQ} \quad \text{Ra.rq,Rb.rq,Rc.wq} \quad \text{!Operate format}
\]

\[
\text{ADDQ} \quad \text{Ra.rq,#b.ib,Rc.wq} \quad \text{!Operate format}
\]

Operation:

\[
\text{Rc} \leftarrow \text{Ra} + \text{Rb}
\]

Exceptions:

Integer Overflow

Instruction mnemonics:

ADDQ Add Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is added to register Rb or a literal, and the 64-bit sum is written to Rc. On overflow, the least significant 64 bits of the true result are written to the destination register.

The unsigned compare instructions can be used to generate carry. After adding two values, if the sum is less unsigned than either one of the inputs, there was a carry out of the most significant bit.
4.4.4 Scaled Quadword Add

Format:

\[
\begin{align*}
SxADDQ & \quad Ra.rq,Rb.rq,Rc.wq \\
SxADDQ & \quad Ra.rq,#b.ib,Rc.wq
\end{align*}
\]

!Operate format

Operation:

\[
\text{CASE} \\
\quad S4ADDQ: \quad Rc \leftarrow \text{LEFT\_SHIFT}(Rav,2) + Rbv \\
\quad S8ADDQ: \quad Rc \leftarrow \text{LEFT\_SHIFT}(Rav,3) + Rbv \\
\text{ENDCASE}
\]

Exceptions:

None

Instruction mnemonics:

\[
\begin{align*}
S4ADDQ & \quad \text{Scaled Add Quadword by 4} \\
S8ADDQ & \quad \text{Scaled Add Quadword by 8}
\end{align*}
\]

Qualifiers:

None

Description:

Register Ra is scaled by 4 (for S4ADDQ) or 8 (for S8ADDQ) and is added to register Rb or a literal, and the 64-bit sum is written to Rc.

On overflow, the least significant 64 bits of the true result are written to the destination register.
4.4.5 Integer Signed Compare

Format:

```
CMPxx   Ra.rq,Rb.rq,Rc.wq  !Operate format
CMPxx   Ra.rq,#b.ib,Rc.wq !Operate format
```

Operation:

```
IF Rav SIGNED_RELATION Rbv THEN
   Rc ← 1
ELSE
   Rc ← 0
```

Exceptions:

None

Instruction mnemonics:

- CMPEQ Compare Signed Quadword Equal
- CMPLT Compare Signed Quadword Less Than
- CMPLE Compare Signed Quadword Less Than or Equal
- CMPLT Compare Signed Quadword Less Than

Qualifiers:

None

Description:

Register Ra is compared to Register Rb or a literal. If the specified relationship is true, the value one is written to register Rc; otherwise, zero is written to Rc.

Notes:

- Compare Less Than A,B is the same as Compare Greater Than B,A; Compare Less Than or Equal A,B is the same as Compare Greater Than or Equal B,A. Therefore, only the less-than operations are included.
4.4.6 Integer Unsigned Compare

Format:

\[
\begin{align*}
\text{CMPU}xx & \quad \text{Ra rq, Rb rq, Rc wq} \quad !\text{Operate format} \\
\text{CMPU}xx & \quad \text{Ra rq, #b ib, Rc wq} \quad !\text{Operate format}
\end{align*}
\]

Operation:

\[
\begin{align*}
\text{IF Ra}_v & \ \text{UNSIGNED\_RELATION Rb}_v \ \text{THEN} \\
& \quad \text{Rc} \leftarrow 1 \\
\text{ELSE} & \\
& \quad \text{Rc} \leftarrow 0
\end{align*}
\]

Exceptions:

None

Instruction mnemonics:

- CMPULE Compare Unsigned Quadword Less Than or Equal
- CMPULT Compare Unsigned Quadword Less Than

Qualifiers:

None

Description:

Register Ra is compared to Register Rb or a literal. If the specified relationship is true, the value one is written to register Rc; otherwise, zero is written to Rc.
4.4.7 Count Leading Zero

Format:

\[
\text{CTLZ Rb.rq,Rc.wq} ! \text{ Operate format}
\]

Operation:

\[
\text{temp} = 0 \\
\text{FOR } i \text{ FROM 63 DOWN TO 0} \\
\quad \text{IF } \{ \text{Rbv}<i> \text{ EQ 1 } \} \text{ THEN BREAK} \\
\quad \text{temp} = \text{temp} + 1 \\
\text{END} \\
\text{Rc}<6:0> \leftarrow \text{temp}<6:0> \\
\text{Rc}<63:7> \leftarrow 0
\]

Exceptions:

None

Instruction mnemonics:

\[
\text{CTLZ Count Leading Zero}
\]

Qualifiers:

None

Description:

The number of leading zeros in Rb, starting at the most significant bit position, is written to Rc. Ra must be R31.
4.4.8 Count Population

Format:

CTPOP     Rb.rq,Rc.wq                   ! Operate format

Operation:

    temp = 0
    FOR i FROM 0 TO 63
        IF { Rbv<i> EQ 1 } THEN temp = temp + 1
    END
    Rc<6:0> ← temp<6:0>
    Rc<63:7> ← 0

Exceptions:

    None

Instruction mnemonics:

    CTPPOP  Count Population

Qualifiers:

    None

Description:

    The number of ones in Rb is written to Rc. Ra must be R31.
4.4.9 Count Trailing Zero

Format:

CTTZ Rb.rq,Rc.wq ! Operate format

Operation:

\[
\text{temp} = 0 \\
\text{FOR } i \text{ FROM 0 TO 63} \\
\quad \text{IF } \{ \text{Rb}v<i> \text{ EQ 1} \} \text{ THEN BREAK} \\
\quad \text{temp} = \text{temp} + 1 \\
\text{END} \\
\text{Rc}<6:0> \leftarrow \text{temp}<6:0> \\
\text{Rc}<63:7> \leftarrow 0
\]

Exceptions:

None

Instruction mnemonics:

CTPOP Count Trailing Zero

Qualifiers:

None

Description:

The number of trailing zeros in Rb, starting at the least significant bit position, is written to Rc. Ra must be R31.
4.4.10 Longword Multiply

Format:

MULL Ra.rl,Rb.rl,Rc.wq !Operate format
MULL Ra.rl,#b.ib,Rc.wq !Operate format

Operation:

\[ R_c \leftarrow \text{SEXT} \left( (R_a \times R_b)_{31:0} \right) \]

Exceptions:

Integer Overflow

Instruction mnemonics:

MULL Multiply Longword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is multiplied by register Rb or a literal, and the sign-extended 32-bit product is written to Rc.

The high 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit product. Overflow detection is based on the longword product \( R_a<31:0> \times R_b<31:0> \). On overflow, the proper sign extension of the least significant 32 bits of the true result are written to the destination register.

The MULQ instruction can be used to return the full 64-bit product.
4.4.11 Quadword Multiply

Format:

\[
\begin{align*}
\text{MULQ} & \quad \text{Ra.rq,Rb.rq,Rc.wq} & \text{!Operate format} \\
\text{MULQ} & \quad \text{Ra.Rq,#b.ib,Rc.wq} & \text{!Operate format}
\end{align*}
\]

Operation:

\[
Rc \leftarrow Ra \times Rb
\]

Exceptions:

Integer Overflow

Instruction mnemonics:

MULQ Multiply Quadword

Qualifiers:

Integer Overflow Enable (/V)

Description:

Register Ra is multiplied by register Rb or a literal, and the 64-bit product is written to register Rc. Overflow detection is based on considering the operands and the result as signed quantities. On overflow, the least significant 64 bits of the true result are written to the destination register.

The UMULH instruction can be used to generate the upper 64 bits of the 128-bit result when an overflow occurs.
4.4.12 Unsigned Quadword Multiply High

Format:

UMULH Ra.rq,Rb.rq,Rc.wq !Operate format
UMULH Ra.Rq,#b.ib,Rc.wq !Operate format

Operation:

\[ \text{Rc} \leftarrow \{\text{Rav} \times \text{U Rbv}\}_{127:64} \]

Exceptions:

None

Instruction mnemonics:

UMULH Unsigned Multiply Quadword High

Qualifiers:

None

Description:

Register Ra and Rb or a literal are multiplied as unsigned numbers to produce a 128-bit result. The high-order 64-bits are written to register Rc.

The UMULH instruction can be used to generate the upper 64 bits of a 128-bit result as follows:

Ra and Rb are unsigned: result of UMULH
Ra and Rb are signed: \( (\text{result of UMULH}) - \text{Ra}_{63} \times \text{Rb} - \text{Rb}_{63} \times \text{Ra} \)

The MULQ instruction gives the low 64 bits of the result in either case.
4.4.13 Longword Subtract

Format:

```
SUBL    Ra.rl,Rb.rl,Rc.wq   !Operate format
SUBL    Ra.rl,#b.ib,Rc.wq   !Operate format
```

Operation:

```
Rc ← SEXT ((Rav − Rbv)<31:0>)
```

Exceptions:

- Integer Overflow

Instruction mnemonics:

```
SUBL    Subtract Longword
```

Qualifiers:

- Integer Overflow Enable (/V)

Description:

Register Rb or a literal is subtracted from register Ra, and the sign-extended 32-bit difference is written to Rc.

The high 32 bits of Ra and Rb are ignored. Rc is a proper sign extension of the truncated 32-bit difference. Overflow detection is based on the longword difference Rav<31:0>− Rbv<31:0>.
4.4.14 Scaled Longword Subtract

Format:

\[
\begin{align*}
\text{SxSUBL} & \quad \text{Ra.rl,Rb.rl,Rc.wq} \quad \text{!Operate format} \\
\text{SxSUBL} & \quad \text{Ra.rl,#b.ib,Rc.wq} \quad \text{!Operate format}
\end{align*}
\]

Operation:

\[
\text{CASE} \\
\text{S4SUBL:} & \quad \text{Rc} \leftarrow \text{SEXT} \left( \left( \text{LEFT}\_\text{SHIFT}(\text{Rav}, 2) \right) - \text{Rbv} < 31:0 > \right) \\
\text{S8SUBL:} & \quad \text{Rc} \leftarrow \text{SEXT} \left( \left( \text{LEFT}\_\text{SHIFT}(\text{Rav}, 3) \right) - \text{Rbv} < 31:0 > \right) \\
\text{ENDCASE}
\]

Exceptions:

None

Instruction mnemonics:

\[
\begin{align*}
\text{S4SUBL} & \quad \text{Scaled Subtract Longword by 4} \\
\text{S8SUBL} & \quad \text{Scaled Subtract Longword by 8}
\end{align*}
\]

Qualifiers:

None

Description:

Register \( Rb \) or a literal is subtracted from the scaled value of register \( Ra \), which is scaled by 4 (for \( \text{S4SUBL} \)) or 8 (for \( \text{S8SUBL} \)), and the sign-extended 32-bit difference is written to \( Rc \).

The high 32 bits of \( Ra \) and \( Rb \) are ignored. \( Rc \) is a proper sign extension of the truncated 32-bit difference.
4.4.15 Quadword Subtract

Format:

\[
\begin{align*}
\text{SUBQ} & \quad \text{Ra.rq,Rb.rq,Rc.wq} \quad \text{!Operate format} \\
\text{SUBQ} & \quad \text{Ra.rq,#b.ib,Rc.wq} \quad \text{!Operate format}
\end{align*}
\]

Operation:

\[
Rc \leftarrow Ra - Rb
\]

Exceptions:

- Integer Overflow

Instruction mnemonics:

- SUBQ Subtract Quadword

Qualifiers:

- Integer Overflow Enable (V)

Description:

Register Rb or a literal is subtracted from register Ra, and the 64-bit difference is written to register Rc. On overflow, the least significant 64 bits of the true result are written to the destination register.

The unsigned compare instructions can be used to generate borrow. If the minuend (Rav) is less unsigned than the subtrahend (Rbv), there will be a borrow.
4.4.16 Scaled Quadword Subtract

Format:

\[ \text{SxSUBQ} \quad \text{Ra.rq,Rb.rq,Rc.wq} \quad \text{!Operate format} \]
\[ \text{SxSUBQ} \quad \text{Ra.rq,\#b.ib,Rc.wq} \quad \text{!Operate format} \]

Operation:

\[
\text{CASE} \\
\begin{align*}
\text{S4SUBQ:} & \quad \text{Rc} \leftarrow \text{LEFTSHIFT}(\text{Ra}, 2) - \text{Rb}v \\
\text{S8SUBQ:} & \quad \text{Rc} \leftarrow \text{LEFTSHIFT}(\text{Ra}, 3) - \text{Rb}v
\end{align*}
\text{ENDCASE}
\]

Exceptions:

None

Instruction mnemonics:

S4SUBQ  Scaled Subtract Quadword by 4
S8SUBQ  Scaled Subtract Quadword by 8

Qualifiers:

None

Description:

Register Rb or a literal is subtracted from the scaled value of register Ra, which is scaled by 4 (for S4SUBQ) or 8 (for S8SUBQ), and the 64-bit difference is written to Rc.
4.5 Logical and Shift Instructions

The logical instructions perform quadword Boolean operations. The conditional move integer instructions perform conditionals without a branch. The shift instructions perform left and right logical shift and right arithmetic shift. These are summarized in Table 4–6.

Table 4–6: Logical and Shift Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Logical Product</td>
</tr>
<tr>
<td>BIC</td>
<td>Logical Product with Complement</td>
</tr>
<tr>
<td>BIS</td>
<td>Logical Sum (OR)</td>
</tr>
<tr>
<td>EQV</td>
<td>Logical Equivalence (XORNOT)</td>
</tr>
<tr>
<td>ORNOT</td>
<td>Logical Sum with Complement</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical Difference</td>
</tr>
<tr>
<td>CMOVxx</td>
<td>Conditional Move Integer</td>
</tr>
<tr>
<td>SLL</td>
<td>Shift Left Logical</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Right Arithmetic</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Right Logical</td>
</tr>
</tbody>
</table>

Software Note:

There is no arithmetic left shift instruction. Where an arithmetic left shift would be used, a logical shift will do. For multiplying by a small power of two in address computations, logical left shift is acceptable.

Integer multiply should be used to perform an arithmetic left shift with overflow checking.

Bit field extracts can be done with two logical shifts. Sign extension can be done with left logical shift and a right arithmetic shift.
4.5.1 Logical Functions

Format:

```
mnemonic Ra.rq,Rb.rq,Rc.wq !Operate format
mnemonic Ra.rq,#b.ib,Rc.wq !Operate format
```

Operation:

```
Rc ← Rav AND Rbv  !AND
Rc ← Rav OR Rbv   !BIS
Rc ← Rav XOR Rbv  !XOR
Rc ← Rav AND {NOT Rbv} !BIC
Rc ← Rav OR {NOT Rbv} !ORNOT
Rc ← Rav XOR {NOT Rbv} !EQV
```

Exceptions:

None

Instruction mnemonics:

- **AND** Logical Product
- **BIC** Logical Product with Complement
- **BIS** Logical Sum (OR)
- **EQV** Logical Equivalence (XORNOT)
- **ORNOT** Logical Sum with Complement
- **XOR** Logical Difference

Qualifiers:

None

Description:

These instructions perform the designated Boolean function between register Ra and register Rb or a literal. The result is written to register Rc.

The “NOT” function can be performed by doing an ORNOT with zero (Ra = R31).
4.5.2 Conditional Move Integer

Format:

CMOVxx Ra.rq,Rb.rq,Rc.wq !Operate format
CMOVxx Ra.rq,#b.ib,Rc.wq !Operate format

Operation:

IF TEST(Rav, Condition_based_on_Opcode) THEN
    Rc ← Rbv

Exceptions:

None

Instruction mnemonics:

CMOVEQ CMOVE if Register Equal to Zero
CMOVGE CMOVE if Register Greater Than or Equal to Zero
CMOVGT CMOVE if Register Greater Than Zero
CMOVLBC CMOVE if Register Low Bit Clear
CMOVLBS CMOVE if Register Low Bit Set
CMOVLE CMOVE if Register Less Than or Equal to Zero
CMOVLT CMOVE if Register Less Than Zero
CMOVNE CMOVE if Register Not Equal to Zero

Qualifiers:

None

Description:

Register Ra is tested. If the specified relationship is true, the value Rbv is written to register Rc.
Notes:
Except that it is likely in many implementations to be substantially faster, the instruction:

        CMOVEQ Ra,Rb,Rc

is exactly equivalent to:

        BNE  Ra,label
        OR   Rb,Rb,Rc

label:  ...

For example, a branchless sequence for:

        R1=MAX(R1,R2)

is:

        CMPLT  R1,R2,R3    ! R3=1 if R1<R2
        CMOVNE R3,R2,R1    ! Move R2 to R1 if R1<R2
4.5.3 Shift Logical

Format:

```
SxL             Ra.rq,Rb.rq,Rc.wq          !Operate format
SxL             Ra.rq,#b.ib,Rc.wq         !Operate format
```

Operation:

```
Rc ← LEFT_SHIFT(Rav, Rbv<5:0>) !SLL
Rc ← RIGHT_SHIFT(Rav, Rbv<5:0>) !SRL
```

Exceptions:

None

Instruction mnemonics:

```
SLL     Shift Left Logical
SRL     Shift Right Logical
```

Qualifiers:

None

Description:

Register Ra is shifted logically left or right 0 to 63 bits by the count in register Rb or a literal. The result is written to register Rc. Zero bits are propagated into the vacated bit positions.
4.5.4 Shift Arithmetic

Format:

SRA Ra.rq,Rb.rq,Rc.wq !Operate format
SRA Ra.rq,#b.ib,Rc.wq !Operate format

Operation:

\[ R_c \leftarrow \text{ARITH_RIGHT_SHIFT}(R_{av}, R_{bv<5:0>}) \]

Exceptions:

None

Instruction mnemonics:

SRA Shift Right Arithmetic

Qualifiers:

None

Description:

Register Ra is right shifted arithmetically 0 to 63 bits by the count in register Rb or a literal. The result is written to register Rc. The sign bit (Rav<63>) is propagated into the vacated bit positions.
4.6 Byte-Manipulation Instructions

Along with the LDBU and STB instructions for loading and storing byte datum between a register and memory, Alpha provides the following instructions for operating on byte operands within registers. These instructions allow full-width memory accesses in the load/store instructions combined with powerful in-register byte manipulation.

The instructions are summarized in Table 4–7.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPBGE</td>
<td>Compare Byte</td>
</tr>
<tr>
<td>EXTBL</td>
<td>Extract Byte Low</td>
</tr>
<tr>
<td>EXTWL</td>
<td>Extract Word Low</td>
</tr>
<tr>
<td>EXTLH</td>
<td>Extract Longword Low</td>
</tr>
<tr>
<td>EXTQL</td>
<td>Extract Quadword Low</td>
</tr>
<tr>
<td>EXTWH</td>
<td>Extract Word High</td>
</tr>
<tr>
<td>EXTLH</td>
<td>Extract Longword High</td>
</tr>
<tr>
<td>EXTQH</td>
<td>Extract Quadword High</td>
</tr>
<tr>
<td>INSBL</td>
<td>Insert Byte Low</td>
</tr>
<tr>
<td>INSWL</td>
<td>Insert Word Low</td>
</tr>
<tr>
<td>INSLL</td>
<td>Insert Longword Low</td>
</tr>
<tr>
<td>INSQL</td>
<td>Insert Quadword Low</td>
</tr>
<tr>
<td>INSWH</td>
<td>Insert Word High</td>
</tr>
<tr>
<td>INSLH</td>
<td>Insert Longword High</td>
</tr>
<tr>
<td>INSQH</td>
<td>Insert Quadword High</td>
</tr>
<tr>
<td>MSKBL</td>
<td>Mask Byte Low</td>
</tr>
<tr>
<td>MSKWL</td>
<td>Mask Word Low</td>
</tr>
<tr>
<td>MSKLL</td>
<td>Mask Longword Low</td>
</tr>
<tr>
<td>MSKQL</td>
<td>Mask Quadword Low</td>
</tr>
<tr>
<td>MSKWH</td>
<td>Mask Word High</td>
</tr>
<tr>
<td>MSKLH</td>
<td>Mask Longword High</td>
</tr>
<tr>
<td>MSKQH</td>
<td>Mask Quadword High</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
</tr>
<tr>
<td>SEXTB</td>
<td>Sign extend byte</td>
</tr>
<tr>
<td>SEXTW</td>
<td>Sign extend word</td>
</tr>
<tr>
<td>ZAP</td>
<td>Zero Bytes</td>
</tr>
<tr>
<td>ZAPNOT</td>
<td>Zero Bytes Not</td>
</tr>
</tbody>
</table>
4.6.1 Compare Byte

Format:

```
CMPBGE Ra.rq,Rb.rq,Rc.wq   !Operate format
CMPBGE Ra.rq,#b.ib,Rc.wq   !Operate format
```

Operation:

```
FOR i FROM 0 TO 7
    temp<8:0> ← {0 || Rav<i*8+7:i*8>} +
                 {0 || NOT Rbv<i*8+7:i*8>} + 1
    Rc<i>   ← temp<8>
END
Rc<63:8> ← 0
```

Exceptions:

None

Instruction mnemonics:

```
CMPBGE  Compare Byte
```

Qualifiers:

None

Description:

CMPBGE does eight parallel unsigned byte comparisons between corresponding bytes of Rav and Rbv, storing the eight results in the low eight bits of Rc. The high 56 bits of Rc are set to zero. Bit 0 of Rc corresponds to byte 0, bit 1 of Rc corresponds to byte 1, and so forth. A result bit is set in Rc if the corresponding byte of Rav is greater than or equal to Rbv (unsigned).
Notes:
The result of CMPBGE can be used as an input to ZAP and ZAPNOT.

To scan for a byte of zeros in a character string:

<initialize R1 to aligned QW address of string>

LOOP:
LDQ R2,0(R1)    ; Pick up 8 bytes
LDA R1,8(R1)    ; Increment string pointer
CMPBGE R31,R2,R3 ; If NO bytes of zero, R3<7:0>=0
BEQ R3,LOOP     ; Loop if no terminator byte found
...             ; At this point, R3 can be used to
determine which byte terminated

To compare two character strings for greater/less:

<initialize R1 to aligned QW address of string1>
<initialize R2 to aligned QW address of string2>

LOOP:
LDQ R3,0(R1)    ; Pick up 8 bytes of string1
LDA R1,8(R1)    ; Increment string1 pointer
LDQ R4,0(R2)    ; Pick up 8 bytes of string2
LDA R2,8(R2)    ; Increment string2 pointer
XOR R3,R4,R5    ; Test for all equal bytes
BEQ R5,LOOP     ; Loop if all equal
CMPBGE R31,R5,R5 ; At this point, R5 can be used to
determine the first not-equal
; byte position.

To range-check a string of characters in R1 for ‘0’..‘9’:

LDQ R2,lit0s    ; Pick up 8 bytes of the character
; BELOW ‘0’ ‘////////’
LDQ R3,lit9s    ; Pick up 8 bytes of the character
; ABOVE ‘9’ ‘::::::::’
CMPBGE R2,R1,R4 ; Some R4<i>=1 if character is LT ‘0’
CMPBGE R1,R3,R5 ; Some R5<i>=1 if character is GT ‘9’
BNE R4,ERROR    ; Branch if some char too low
BNE R5,ERROR    ; Branch if some char too high
4.6.2 Extract Byte

Format:

\[
\begin{align*}
\text{EXTxx} &\quad \text{Ra.rq, Rb.rq, Rc.wq} & \text{!Operate format} \\
\text{EXTxx} &\quad \text{Ra.rq, #b.ib, Rc.wq} & \text{!Operate format}
\end{align*}
\]

Operation:

CASE

\[
\begin{align*}
\text{big_endian_data: } &\quad Rbv’ \leftarrow Rbv \text{ XOR } 111 2 \quad \text{big_endian_data: } Rbv’ \leftarrow Rbv
\end{align*}
\]

ENDCASE

CASE

\[
\begin{align*}
\text{EXTBL: } &\quad \text{byte_mask} \leftarrow 0000 0001_2 \\
\text{EXTWx: } &\quad \text{byte_mask} \leftarrow 0000 0011_2 \\
\text{EXTLx: } &\quad \text{byte_mask} \leftarrow 0000 1111_2 \\
\text{EXTQx: } &\quad \text{byte_mask} \leftarrow 1111 1111_2
\end{align*}
\]

ENDCASE

CASE

\[
\begin{align*}
\text{EXTxL: } &\quad \text{byte_loc} \leftarrow Rbv’<2:0>*8 \\
\text{temp} &\quad \leftarrow \text{RIGHT_SHIFT}(Rav, \text{byte_loc}<5:0>) \\
\text{Rc} &\quad \leftarrow \text{BYTE_ZAP}(\text{temp, NOT(byte_mask)})
\end{align*}
\]

\[
\begin{align*}
\text{EXTxH: } &\quad \text{byte_loc} \leftarrow 64 - Rbv’<2:0>*8 \\
\text{temp} &\quad \leftarrow \text{LEFT_SHIFT}(Rav, \text{byte_loc}<5:0>) \\
\text{Rc} &\quad \leftarrow \text{BYTE_ZAP}(\text{temp, NOT(byte_mask)})
\end{align*}
\]

ENDCASE

Exceptions:

None

Instruction mnemonics:

- **EXTBL** Extract Byte Low
- **EXTWL** Extract Word Low
- **EXTLL** Extract Longword Low
- **EXTQL** Extract Quadword Low
- **EXTWH** Extract Word High
- **EXTLH** Extract Longword High

---

4–52 Alpha Architecture Handbook
EXTQH      Extract Quadword High

Qualifiers:
None

Description:
EXTxL shifts register Ra right by 0 to 7 bytes, inserts zeros into vacated bit positions, and then extracts 1, 2, 4, or 8 bytes into register Rc. EXTxH shifts register Ra left by 0 to 7 bytes, inserts zeros into vacated bit positions, and then extracts 2, 4, or 8 bytes into register Rc. The number of bytes to shift is specified by Rbv’<2:0>. The number of bytes to extract is specified in the function code. Remaining bytes are filled with zeros.

Notes:
The comments in the examples below assume that the effective address (ea) of X(R11) is such that (ea mod 8) = 5, the value of the aligned quadword containing X(R11) is CBAx xxxx, and the value of the aligned quadword containing X+7(R11) is yyyH GFED, and the datum is little-endian.

The examples below are the most general case unless otherwise noted; if more information is known about the value or intended alignment of X, shorter sequences can be used.

The intended sequence for loading a quadword from unaligned address X(R11) is:

LDQ_U R1,X(R11) ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U R2,X+7(R11) ; Ignores va<2:0>, R2 = yyyH GFED
LDA R3,X(R11) ; R3<2:0> = (X mod 8) = 5
EXTQL R1,R3,R1 ; R1 = 0000 0CBA
EXTQH R2,R3,R2 ; R2 = HGFE D000
OR R2,R1,R1 ; R1 = HGFE DCBA

The intended sequence for loading and zero-extending a longword from unaligned address X is:

LDQ_U R1,X(R11) ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U R2,X+3(R11) ; Ignores va<2:0>, R2 = yyyy yyyD
LDA R3,X(R11) ; R3<2:0> = (X mod 8) = 5
EXTLL R1,R3,R1 ; R1 = 0000 0CBA
EXTLH R2,R3,R2 ; R2 = 0000 D000
OR R2,R1,R1 ; R1 = 0000 DCBA

The intended sequence for loading and sign-extending a longword from unaligned address X is:
LDQ_U R1, X(R11) ; Ignores va<2:0>, R1 = CBAx xxxx
LDQ_U R2, X+3(R11) ; Ignores va<2:0>, R2 = yyyy yyyD
LDA R3, X(R11) ; R3<2:0> = (X mod 8) = 5
EXTLL R1, R3, R1 ; R1 = 0000 0CBA
EXTLH R2, R3, R2 ; R2 = 0000 DD00
OR R2, R1, R1 ; R1 = 0000 DCBA
ADDL R31, R1, R1 ; R1 = ssss DCBA

For software that is not designed to use the BWX extension1, the intended sequence
for loading and zero-extending a word from unaligned address X is:

LDQ_U R1, X(R11) ; Ignores va<2:0>, R1 = yBAx xxxx
LDQ_U R2, X+1(R11) ; Ignores va<2:0>, R2 = yBAx xxxx
LDA R3, X(R11) ; R3<2:0> = (X mod 8) = 5
EXTWL R1, R3, R1 ; R1 = 0000 00BA
EXTWH R2, R3, R2 ; R2 = 0000 0000
OR R2, R1, R1 ; R1 = 0000 00BA

For software that is not designed to use the BWX extension1, the intended sequence
for loading and sign-extending a word from unaligned address X is:

LDQ_U R1, X(R11) ; Ignores va<2:0>, R1 = yBAx xxxx
LDQ_U R2, X+1(R11) ; Ignores va<2:0>, R2 = yBAx xxxx
LDA R3, X(R11) ; R3<2:0> = (X mod 8) = 5
EXTQL R1, R3, R1 ; R1 = 0000 000y
EXTQH R2, R3, R2 ; R2 = BAxx xxx0
OR R2, R1, R1 ; R1 = BAxx xxxy
SRA R1, #48, R1 ; R1 = ssss ssBA

For software that is not designed to use the BWX extension1, the intended sequence
for loading and zero-extending a byte from address X is:

LDQ_U R1, X(R11) ; Ignores va<2:0>, R1 = yyAx xxxx
LDA R3, X(R11) ; R3<2:0> = (X mod 8) = 5
EXTBL R1, R3, R1 ; R1 = 0000 000A

For software that is not designed to use the BWX extension1, the intended sequence
for loading and sign-extending a byte from address X is:

LDQ_U R1, X(R11) ; Ignores va<2:0>, R1 = yyAx xxxx
LDA R3, X+1(R11) ; R3<2:0> = (X + 1) mod 8, i.e.,
; convert byte position within
; quadword to one-origin based
; EXTQH R1, R3, R1 ; Places the desired byte into byte 7
; of R1.final by left shifting
; R1.initial by (8 - R3<2:0>) byte
; positions
SRA R1, #56, R1 ; Arithmetic Shift of byte 7 down
; into byte 0,

Optimized examples:
Assume that a word fetch is needed from 10(R3), where R3 is intended to contain
a longword-aligned address. The optimized sequences below take advantage of the

1 See the Preface and Appendix D for information about the BWX extension.
known constant offset, and the longword alignment (hence a single aligned longword contains the entire word). The sequences generate a Data Alignment Fault if R3 does not contain a longword-aligned address.

For software that is not designed to use the BWX extension, the intended sequence for loading and zero-extending an aligned word from 10(R3) is:

```assembly
LDL R1,8(R3) ; R1 = ssss BAxx
EXTWL R1,#2,R1 ; R1 = 0000 00BA
```

For software that is not designed to use the BWX extension, the intended sequence for loading and sign-extending an aligned word from 10(R3) is:

```assembly
LDL R1,8(R3) ; R1 = ssss BAxx
SRA R1,#16,R1 ; R1 = ssss ssBA
```

**Big-endian examples:**

For software that is not designed to use the BWX extension, the intended sequence for loading and zero-extending a byte from address X is:

```assembly
LDQ_U R1,X(R11) ; Ignores va<2:0>, R1 = xxxx xAyy
LDA R3,X(R11) ; R3<2:0> = 5, shift will be 2 bytes
EXTBL R1,R3,R1 ; R1 = 0000 000A
```

The intended sequence for loading a quadword from unaligned address X(R11) is:

```assembly
LDQ_U R1,X(R11) ; Ignores va<2:0>, R1 = xxxxxABC
LDQ_U R2,X+7(R11) ; Ignores va<2:0>, R2 = DEFGHyyy
LDA R3,X+7(R11) ; R3<2:0> = 4, shift will be 3 bytes
EXTQH R1,R3,R1 ; R1 = ABC0 0000
EXTQL R2,R3,R2 ; R2 = 000D EFGH
OR R1,R2,R1 ; R1 = ABCD EFGH
```

Note that the address in the LDA instruction for big-endian quadwords is X+7, for longwords is X+3, and for words is X+1; for little-endian, these are all just X. Also note that the EXTQH and EXTQL instructions are reversed with respect to the little-endian sequence.

---

1 See the Preface and Appendix D for information about the BWX extension.
4.6.3 Byte Insert

Format:

<table>
<thead>
<tr>
<th>Format</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSxx</td>
<td>Ra.rq,Rb.rq,Rc.wq !Operate format</td>
</tr>
<tr>
<td>INSxx</td>
<td>Ra.rq,#b.ib,Rc.wq !Operate format</td>
</tr>
</tbody>
</table>

Operation:

```
CASE
  big_endian_data: Rbv' ← Rbv XOR 1112
  little_endian_data: Rbv' ← Rbv
ENDCASE
CASE
  INSBL: byte_mask ← 0000 0000 0000 00012
  INSWx: byte_mask ← 0000 0000 0000 00112
  INSLx: byte_mask ← 0000 0000 0000 11112
  INSQL: byte_mask ← 0000 0000 1111 11112
ENDCASE
byte_mask ← LEFT_SHIFT(byte_mask, Rbv'<2:0>)
CASE
  INSxL: byte_loc ← Rbv'<2:0>*8
    temp ← LEFT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask<7:0>))
  INSxH: byte_loc ← 64 - Rbv'<2:0>*8
    temp ← RIGHT_SHIFT(Rav, byte_loc<5:0>)
    Rc ← BYTE_ZAP(temp, NOT(byte_mask<15:8>))
ENDCASE
```

Exceptions:

None

Instruction mnemonics:

- INSBL  Insert Byte Low
- INSWL  Insert Word Low
- INSLL  Insert Longword Low
- INSQL  Insert Quadword Low
- INSWH  Insert Word High
- INSLH  Insert Longword High
INSQH  Insert Quadword High

Qualifiers:
None

Description:
INSxL and INSxH shift bytes from register Ra and insert them into a field of zeros, storing the result in register Rc. Register Rbv'2:0 selects the shift amount, and the function code selects the maximum field width: 1, 2, 4, or 8 bytes. The instructions can generate a byte, word, longword, or quadword datum that is spread across two registers at an arbitrary byte alignment.
4.6.4 Byte Mask

Format:

\[
\begin{align*}
\text{MSKxx} & \quad \text{Ra.rq, Rb.rq, Rc.wq} \quad \text{!Operate format} \\
\text{MSKxx} & \quad \text{Ra.rq, #b.ib, Rc.wq} \quad \text{!Operate format}
\end{align*}
\]

Operation:

\[
\begin{align*}
\text{CASE} \\
\text{big_endian_data:} & \quad Rbv' \leftarrow Rbv \ XOR \ 111_2 \\
\text{little_endian_data:} & \quad Rbv' \leftarrow Rbv \\
\text{ENDCASE} \\
\text{CASE} \\
\text{MSKBL:} & \quad \text{byte_mask} \leftarrow 0000 0000 0000 0001_2 \\
\text{MSKWx:} & \quad \text{byte_mask} \leftarrow 0000 0000 0000 0011_2 \\
\text{MSKLx:} & \quad \text{byte_mask} \leftarrow 0000 0000 0000 1111_2 \\
\text{MSKQx:} & \quad \text{byte_mask} \leftarrow 0000 0000 1111 1111_2 \\
\text{ENDCASE} \\
\text{byte_mask} & \leftarrow \text{LEFT_SHIFT} (\text{byte_mask, Rbv'}<2:0>) \\
\text{CASE} \\
\text{MSKxL:} & \quad \text{Rc} \leftarrow \text{BYTE_ZAP} (\text{Rav, byte_mask}<7:0>) \\
\text{MSKxH:} & \quad \text{Rc} \leftarrow \text{BYTE_ZAP} (\text{Rav, byte_mask}<15:8>) \\
\text{ENDCASE}
\end{align*}
\]

Exceptions:

None

Instruction mnemonics:

- MSKBL  Mask Byte Low
- MSKWL  Mask Word Low
- MSKLL  Mask Longword Low
- MSKQL  Mask Quadword Low
- MSKWH  Mask Word High
- MSKLG  Mask Longword High
- MSKQL  Mask Quadword High
Qualifiers:

None

Description:

MSKxL and MSKxH set selected bytes of register Ra to zero, storing the result in register Rc. Register Rbv’<2:0> selects the starting position of the field of zero bytes, and the function code selects the maximum width: 1, 2, 4, or 8 bytes. The instructions generate a byte, word, longword, or quadword field of zeros that can spread across two registers at an arbitrary byte alignment.

Notes:

The comments in the examples below assume that the effective address (ea) of X(R11) is such that (ea mod 8) = 5, the value of the aligned quadword containing X(R11) is CBAx xxxx, the value of the aligned quadword containing X+7(R11) is yyyH GFED, the value to be stored from R5 is HGFE DCBA, and the datum is little-endian. Slight modifications similar to those in Section 4.6.2 apply to big-endian data.

The examples below are the most general case; if more information is known about the value or intended alignment of X, shorter sequences can be used.

The intended sequence for storing an unaligned quadword R5 at address X(R11) is:

```
LDA   R6, X(R11)       ; R6<2:0> = (X mod 8) = 5
LDQ_U R2, X+7(R11)    ; Ignores va<2:0>, R2 = yyyH GFED
LDQ_U R1, X(R11)      ; Ignores va<2:0>, R1 = CBAx xxxx
INSQH R5, R6, R4      ; R4 = 000H GFED
INSQL R5, R6, R3      ; R3 = CBA0 0000
MSKQH R2, R6, R2      ; R2 = yyy0 0000
MSKQL R1, R6, R1      ; R1 = 000x xxxx
OR    R2, R4, R2      ; R2 = yyyH GFED
OR    R1, R3, R1      ; R1 = CBAx xxxx
STQ_U R2, X+7(R11)    ; Must store high then low for
STQ_U R1, X(R11)      ; degenerate case of aligned QW
```

The intended sequence for storing an unaligned longword R5 at X is:

```
LDA   R6, X(R11)       ; R6<2:0> = (X mod 8) = 5
LDQ_U R2, X+3(R11)    ; Ignores va<2:0>, R2 = yyyy yyyD
LDQ_U R1, X(R11)      ; Ignores va<2:0>, R1 = CBAx xxxx
INSLH R5, R6, R4      ; R4 = 0000 000D
INSLL R5, R6, R3      ; R3 = CBA0 0000
MSKLH R2, R6, R2      ; R2 = yyyy yyy0
MSKLL R1, R6, R1      ; R1 = 000x xxxx
OR    R2, R4, R2      ; R2 = yyyy yyyD
OR    R1, R3, R1      ; R1 = CBAx xxxx
STQ_U R2, X+3(R11)    ; Must store high then low for
STQ_U R1, X(R11)      ; degenerate case of aligned
```
For software that is not designed to use the BWX extension, the intended sequence for storing an unaligned word R5 at X is:

LDA R6,X(R11) ; R6<2:0> = (X mod 8) = 5
LDQ_U R2,X+1(R11) ; Ignores va<2:0>, R2 = yBAx xxxx
LDQ_U R1,X(R11) ; Ignores va<2:0>, R1 = yBAx xxxx
INSWH R5,R6,R4 ; R4 = 0000 0000
INSWL R5,R6,R3 ; R3 = 0BA0 0000
MSKWH R2,R6,R2 ; R2 = yBAx xxxx
MSKWL R1,R6,R1 ; R1 = y00x xxxx
OR R2,R4,R2 ; R2 = yBAx xxxx
OR R1,R3,R1 ; R1 = yBAx xxxx
STQ_U R2,X+1(R11) ; Must store high then low for degenerate case of aligned
STQ_U R1,X(R11) ;

For software that is not designed to use the BWX extension, the intended sequence for storing a byte R5 at X is:

LDA R6,X(R11) ; R6<2:0> = (X mod 8) = 5
LDQ_U R1,X(R11) ; Ignores va<2:0>, R1 = yyAx xxxx
INSBL R5,R6,R3 ; R3 = 00A0 0000
MSKBL R1,R6,R1 ; R1 = yy0x xxxx
OR R1,R3,R1 ; R1 = yyAx xxxx
STQ_U R1,X(R11) ;

\(^{1}\) See the Preface and Appendix D for information about the BWX extension.
4.6.5 Sign Extend

Format:

SEXTx    Rb.rq,Rc.wq        !Operate format
SEXTx    #b.ib,Rc.wq        !Operate format

Operation:

CASE
  SEXTB:  Rc ← SEXT(Rbv<07:0>)
  SEXTW:  Rc ← SEXT(Rbv<15:0>)
ENDCASE

Exceptions:

None

Instruction mnemonics:

SEXTB    Sign Extend Byte
SEXTW    Sign Extend Word

Qualifiers:

None

Description:

The byte or word in register Rb is sign-extended to 64 bits and written to register Rc. Ra must be R31.
4.6.6 Zero Bytes

Format:

\[
\begin{align*}
ZAP & \quad Ra rq, Rb rq, Rc wq & \text{!Operate format} \\
ZAP & \quad Ra rq, \#b ib, Rc wq & \text{!Operate format}
\end{align*}
\]

Operation:

\[
\begin{align*}
\text{CASE} \\
\text{ZAP}: & \quad Rc \leftarrow \text{BYTE\_ZAP}(Rav, Rbv<7:0>) \\
\text{ZAPNOT}: & \quad Rc \leftarrow \text{BYTE\_ZAP}(Rav, \text{NOT } Rbv<7:0>) \\
\text{ENDCASE}
\end{align*}
\]

Exceptions:

None

Instruction mnemonics:

\[
\begin{align*}
\text{ZAP} & \quad \text{Zero Bytes} \\
\text{ZAPNOT} & \quad \text{Zero Bytes Not}
\end{align*}
\]

Qualifiers:

None

Description:

ZAP and ZAPNOT set selected bytes of register Ra to zero, and store the result in register Rc. Register Rb<7:0> selects the bytes to be zeroed; bit 0 of Rbv corresponds to byte 0, bit 1 of Rbv corresponds to byte 1, and so on. A result byte is set to zero if the corresponding bit of Rbv is a one for ZAP and a zero for ZAPNOT.
4.7 Floating-Point Instructions

Alpha provides instructions for operating on floating-point operands in each of four data formats:

- F_floating (VAX single)
- G_floating (VAX double, 11-bit exponent)
- S_floating (IEEE single)
- T_floating (IEEE double, 11-bit exponent)

Data conversion instructions are also provided to convert operands between floating-point and quadword integer formats, between double and single floating, and between quadword and longword integers.

Note:

D_floating is a partially supported datatype; no D_floating arithmetic operations are provided in the architecture. For backward compatibility, exact D_floating arithmetic may be provided via software emulation. D_floating “format compatibility,” in which binary files of D_floating numbers may be processed but without the last 3 bits of fraction precision, can be obtained via conversions to G_floating, G arithmetic operations, then conversion back to D_floating.

The choice of data formats is encoded in each instruction. Each instruction also encodes the choice of rounding mode and the choice of trapping mode.

All floating-point operate instructions (that is, not including loads or stores) that yield an F_ or G_floating zero result must materialize a true zero.

4.7.1 Floating-Point Single-Precision Operations

Single-precision values (F_floating or S_floating) are stored in the floating-point registers in canonical form, as subsets of double-precision values, with 11-bit exponents restricted to the corresponding single-precision range, and with the 29 low-order fraction bits restricted to be all zero.


Longword integer values in floating-point registers are stored in bits <63:62,58:29>, with bits <61:59> ignored and zeros in bits <28:0>.

4.7.2 Floating Subsets and Floating Faults

All floating-point operations may take floating disabled faults. Any subsetted floating-point instruction may take an Illegal Instruction Trap. These faults are not explicitly listed in the description of each instruction.

All floating-point loads and stores may take memory management faults (access control violation, translation not valid, fault on read/write, data alignment).
The floating-point enable (FEN) internal processor register (IPR) allows system software to restrict access to the floating-point registers.

If a floating-point instruction is implemented and FEN = 0, attempts to execute the instruction cause a floating disabled fault.

If a floating-point instruction is not implemented, attempts to execute the instruction cause an Illegal Instruction Trap. This rule holds regardless of the value of FEN.

An Alpha implementation may provide both VAX and IEEE floating-point operations, either, or none.

Some floating-point instructions are common to the VAX and IEEE subsets, some are VAX only, and some are IEEE only. These are designated in the descriptions that follow. If either subset is implemented, all the common instructions must be implemented.

An implementation that includes IEEE floating-point may subset the ability to perform rounding to plus infinity and minus infinity. If not implemented, instructions requesting these rounding modes take Illegal Instruction Trap.

An implementation that includes IEEE floating-point may implement any subset of the Trap Disable flags. If a flag is not implemented, it reads as zero and the corresponding trap occurs as usual.

4.7.3 Definitions

The following definitions apply to Alpha floating-point support.

**Alpha finite number**
A floating-point number with a definite, in-range value. Specifically, all numbers in the inclusive ranges –MAX through –MIN, zero, and +MIN through +MAX, where MAX is the largest non-infinite representable floating-point number and MIN is the smallest non-zero representable normalized floating-point number.

For VAX floating-point, finites do not include reserved operands or dirty zeros (this differs from the usual VAX interpretation of dirty zeros as finite). For IEEE floating-point, finites do not include infinites, NaNs, or denormals, but do include minus zero.

**denormal**
An IEEE floating-point bit pattern that represents a number whose magnitude lies between zero and the smallest finite number.

**dirty zero**
A VAX floating-point bit pattern that represents a zero value, but not in true-zero form.

**infinity**
An IEEE floating-point bit pattern that represents plus or minus infinity.
**LSB**
The least significant bit. For a positive representable number \( A \) whose fraction is not all ones, \( A + 1 \) LSB is the next larger representable number, and \( A + 1/2 \) LSB is exactly halfway between \( A \) and the next larger representable number.

**non-finite number**
An IEEE infinity, NaN, denormal number, or a VAX dirty zero or reserved operand.

**Not-a-Number**
An IEEE floating-point bit pattern that represents something other than a number. This comes in two forms: signaling NaNs (for Alpha, those with an initial fraction bit of 0) and quiet NaNs (for Alpha, those with initial fraction bit of 1).

**representable result**
A real number that can be represented exactly as a VAX or IEEE floating-point number, with finite precision and bounded exponent range.

**reserved operand**
A VAX floating-point bit pattern that represents an illegal value.

**trap shadow**
The set of instructions potentially executed after an instruction that signals an arithmetic trap but before the trap is actually taken.

**true result**
The mathematically correct result of an operation, assuming that the input operand values are exact. The true result is typically rounded to the nearest representable result.

**true zero**
The value +0, represented as exactly 64 zeros in a floating-point register.

### 4.7.4 Encodings

Floating-point numbers are represented with three fields: sign, exponent, and fraction. The sign is 1 bit; the exponent is 8, 11, or 15 bits; and the fraction is 23, 52, 55, or 112 bits. Some encodings represent special values:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
<th>VAX Meaning</th>
<th>IEEE Meaning</th>
<th>IEEE Finite</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>All-1's</td>
<td>Non-zero</td>
<td>Finite</td>
<td>Yes</td>
<td>+/-NaN</td>
</tr>
<tr>
<td>x</td>
<td>All-1's</td>
<td>0</td>
<td>Finite</td>
<td>Yes</td>
<td>+/-Infinity</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Non-zero</td>
<td>Dirty zero</td>
<td>No</td>
<td>+/-Denormal</td>
</tr>
</tbody>
</table>
The values of MIN and MAX for each of the five floating-point data formats are:

<table>
<thead>
<tr>
<th>Data Format</th>
<th>MIN</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_floating</td>
<td>(2^{127} \times 0.5) ((2.93873588e-38))</td>
<td>(2^{127} \times (1.0 - 2^{-53})) ((1.7014117e38))</td>
</tr>
<tr>
<td>G_floating</td>
<td>(2^{1023} \times 0.5) ((0.55626846546268004e-308))</td>
<td>(2^{1023} \times (1.0 - 2^{-53})) ((0.8988465743115785407e308))</td>
</tr>
<tr>
<td>S_floating</td>
<td>(2^{126} \times 1.0) ((1.17549435e-38))</td>
<td>(2^{127} \times (2.0 - 2^{-23})) ((3.40282347e38))</td>
</tr>
<tr>
<td>T_floating</td>
<td>(2^{1022} \times 1.0) ((2.2250738585072013e-308))</td>
<td>(2^{1023} \times (2.0 - 2^{-23})) ((1.7976931348623158e308))</td>
</tr>
<tr>
<td>X_floating</td>
<td>(2^{16382} \times 1.0) ((3.36210314311209350626267781732175260e-4932))</td>
<td>(2^{16383} \times (2.0 - 2^{-112})) ((1.8973149535723176508575932662800702e4932))</td>
</tr>
</tbody>
</table>

4.7.5 Floating-Point Rounding Modes

All rounding modes map a true result that is exactly representable to that representable value.

VAX Rounding Modes
For VAX floating-point operations, two rounding modes are provided and are specified in each instruction: normal (biased) rounding and chopped rounding.

Normal VAX rounding maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the larger in absolute value (sometimes called biased rounding away from zero); maps true results \(\geq\) MAX + 1/2 LSB in magnitude to an overflow; maps true results < MIN − 1/2 LSB in magnitude to an underflow.

Chopped VAX rounding maps the true result to the smaller in magnitude of two surrounding representable results; maps true results \(\geq\) MAX + 1 LSB in magnitude to an overflow; maps true results < MIN in magnitude to an underflow.

IEEE Rounding Modes
For IEEE floating-point operations, four rounding modes are provided: normal rounding (unbiased round to nearest), rounding toward minus infinity, round toward zero, and rounding toward plus infinity. The first three can be specified in the
Rounding toward plus infinity can be obtained by setting the Floating-point Control Register (FPCR) to select it and then specifying dynamic rounding mode in the instruction (See Section 4.7.7). Alpha IEEE arithmetic does rounding before detecting overflow/underflow.

Normal IEEE rounding maps the true result to the nearest of two representable results, with true results exactly halfway between mapped to the one whose fraction ends in 0 (sometimes called unbiased rounding to even); maps true results \( \geq \text{MAX} + 1/2 \text{ LSB} \) in magnitude to an overflow; maps true results \( < \text{MIN} - 1/2 \text{ LSB} \) in magnitude to an underflow.

Plus infinity IEEE rounding maps the true result to the larger of two surrounding representable results; maps true results \( > \text{MAX} \) in magnitude to an overflow; maps positive true results \( \leq +\text{MIN} - 1 \text{ LSB} \) to an underflow; and maps negative true results \( > -\text{MIN} \) to an underflow.

Minus infinity IEEE rounding maps the true result to the smaller of two surrounding representable results; maps true results \( > \text{MAX} \) in magnitude to an overflow; maps positive true results \( < +\text{MIN} \) to an underflow; and maps negative true results \( \geq -\text{MIN} + 1 \text{ LSB} \) to an underflow.

Chopped IEEE rounding maps the true result to the smaller in magnitude of two surrounding representable results; maps true results \( \geq \text{MAX} + 1 \text{ LSB} \) in magnitude to an overflow; and maps non-zero true results \( < \text{MIN} \) in magnitude to an underflow.

Dynamic rounding mode uses the IEEE rounding mode selected by the FPCR register and is described in more detail in Section 4.7.7.

The following tables summarize the floating-point rounding modes:

<table>
<thead>
<tr>
<th>VAX Rounding Mode</th>
<th>Instruction Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal rounding</td>
<td>(No modifier)</td>
</tr>
<tr>
<td>Chopped</td>
<td>/C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IEEE Rounding Mode</th>
<th>Instruction Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal rounding</td>
<td>(No modifier)</td>
</tr>
<tr>
<td>Dynamic rounding</td>
<td>/D</td>
</tr>
<tr>
<td>Plus infinity</td>
<td>/D and ensure that FPCR(\text{DYN}) = ‘11’</td>
</tr>
<tr>
<td>Minus infinity</td>
<td>/M</td>
</tr>
<tr>
<td>Chopped</td>
<td>/C</td>
</tr>
</tbody>
</table>
4.7.6 Floating-Point Trapping Modes

There are six exceptions that can be generated by floating-point operate instructions, all signaled by an arithmetic exception trap. These exceptions are:

- Invalid operation
- Division by zero
- Overflow
- Underflow, may be disabled
- Inexact result, may be disabled
- Integer overflow (conversion to integer only), may be disabled

For more detail on the information passed to an arithmetic exception handler, see Part II, Operating Systems.

VAX Trapping Modes

For VAX floating-point operations other than CVTxQ, four trapping modes are provided. They specify software completion and whether traps are enabled for underflow.

For VAX conversions from floating-point to integer, four trapping modes are provided. They specify software completion and whether traps are enabled for integer overflow.

IEEE Trapping Modes

For IEEE floating-point operations other than CVTxQ, four trapping modes are provided. They specify software completion and whether traps are enabled for underflow and inexact results.

For IEEE conversions from floating-point to integer, four trapping modes are provided. They specify software completion, and whether traps are enabled for integer overflow and inexact results.

The modes and instruction notation are:

<table>
<thead>
<tr>
<th>VAX Trap Mode</th>
<th>Instruction Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imprecise, underflow disabled</td>
<td>(No modifier)</td>
</tr>
<tr>
<td>Imprecise, underflow enabled</td>
<td>/U</td>
</tr>
<tr>
<td>Software, underflow disabled</td>
<td>/S</td>
</tr>
<tr>
<td>Software, underflow enabled</td>
<td>/SU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VAX Convert-to-Integer Trap Mode</th>
<th>Instruction Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imprecise, integer overflow disabled</td>
<td>(No modifier)</td>
</tr>
</tbody>
</table>
### VAX Convert-to-Integer Trap Mode

<table>
<thead>
<tr>
<th>Imprecise, integer overflow enabled</th>
<th>/V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software, integer overflow disabled</td>
<td>/S</td>
</tr>
<tr>
<td>Software, integer overflow enabled</td>
<td>/SV</td>
</tr>
</tbody>
</table>

### IEEE Trap Mode

<table>
<thead>
<tr>
<th>Imprecise, unfl disabled, inexact disabled</th>
<th>(No modifier)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imprecise, unfl enabled, inexact disabled</td>
<td>/U</td>
</tr>
<tr>
<td>Software, unfl enabled, inexact disabled</td>
<td>/SU</td>
</tr>
<tr>
<td>Software, unfl enabled, inexact enabled</td>
<td>/SUI</td>
</tr>
</tbody>
</table>

### IEEE Convert-to-Integer Trap Mode

<table>
<thead>
<tr>
<th>Imprecise, int.ovfl disabled, inexact disabled</th>
<th>(No modifier)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imprecise, int.ovfl enabled, inexact disabled</td>
<td>/V</td>
</tr>
<tr>
<td>Software, int.ovfl enabled, inexact disabled</td>
<td>/SV</td>
</tr>
<tr>
<td>Software, int.ovfl enabled, inexact enabled</td>
<td>/SVI</td>
</tr>
</tbody>
</table>

#### 4.7.6.1 Imprecise /Software Completion Trap Modes

Floating-point instructions may be pipelined, and all hardware exceptions are imprecise traps:

- For the floating overflow, divide by zero, and invalid operation exceptions, the trapping instruction may write an UNPREDICTABLE result value.
- The trap PC is an arbitrary number of instructions past the one triggering the trap. The trigger instruction plus all intervening executed instructions are collectively referred to as the trap shadow of the trigger instruction.
- The extent of the trap shadow is bounded only by an EXCB or TRAPB instruction (or the implicit TRAPB within a CALL_PAL instruction).
- Input operand values may have been overwritten in the trap shadow.
- Result values may have been overwritten in the trap shadow.
- An UNPREDICTABLE result value may have been used as an input operand in the trap shadow.
- Additional traps may occur in the trap shadow.
- In general, it is not feasible to fix up the result value or to continue from the trap.
This behavior is ideal for operations on finite operands that give finite results. For programs that deliberately operate outside the overflow/underflow range, or use IEEE NaNs, software assistance is required to complete floating-point operations correctly. This assistance can be provided by a software arithmetic trap handler, plus constraints on the instructions surrounding the trap.

For a trap handler to complete non-finite arithmetic, the conditions described below must hold:

- Conditions 1–3 allow a software trap handler to emulate the trigger instruction with its original input operand values and then to reexecute the rest of the trap shadow.
- Condition 4 prevents memory accesses at UNPREDICTABLE addresses.
- Conditions 5–7 make it possible for a software trap handler to find the trigger instruction via a linear scan backwards from the trap PC.

**Conditions**

1. If the value in a register or memory location is used as input to some instruction in the trap shadow, then either the following condition a or condition b must be met.
   
   a. The register or memory location is not modified by the instruction that uses it or by any subsequent instruction in the trap shadow.
   
   b. The value was produced by an earlier instruction in the trap shadow, and no trapping instruction appears between the producing and consuming instructions.

   Condition a ensures that if the instruction is reexecuted, its inputs are unchanged. If condition a cannot be ensured, then condition b requires that the input values be created and hence valid when reexecution starts at the trigger instruction.

2. If a conditional move (CMOVxx or FCMOVxx) instruction appears in the trap shadow, then the Ra/Fa and Rb/Fb operands of the instruction must satisfy condition 1 above and either the following condition a or condition b must be met.
   
   a. The Ra/Fa operand of the conditional move does not depend on any value produced earlier in the trap shadow by an instruction that might trap.
   
   b. The Rc/Fc operand of the instruction was produced by an earlier instruction in the trap shadow, and no trapping instruction appears between the producing instruction and the conditional move.

   Condition a ensures that the conditional move instruction will be reexecuted with the same condition in Ra/Fa. If condition a cannot be ensured, then condition b requires that the value that might be overlaid is valid when the conditional move is reexecuted.
3. If a value is produced in the trap shadow as the result of a floating-point instruction that might trap, that value may not contribute to any value that is subsequently used in the trap shadow as the input to an integer instruction that has the /V modifier.

4. Within the trap shadow, the computation of the base register for a memory load or store instruction may not involve using the result of an instruction that might generate an UNPREDICTABLE result.

5. Within the trap shadow, if a register is used as the destination register of an instruction that might cause a floating-point exception (and thus set a bit in the software implementation's exception summary), it may not be used as the destination of any other instruction in the trap shadow.

6. The trap shadow may not include any branch instructions.

7. Each floating-point instruction to be completed must be so marked, by specifying the /S software completion modifier. The /S modifier must not be used on any floating-point instruction that is not in a trap shadow that meets these conditions.

Note:
The /S modifier does not affect instruction operation or trap behavior; it is an informational bit passed to a software trap handler. It allows a trap handler to test easily whether an instruction is intended to be completed. (The /S bits of instructions signaling traps are carried into a software implementation's exception summary. The handler may then assume that the other conditions are met without examining the code stream.

If a software trap handler is provided, it must handle the completion of all floating-point operations marked /S that follow the rules above. In effect, one TRAPB instruction per basic block can be used.

4.7.6.2 Invalid Operation (INV) Arithmetic Trap

An invalid operation arithmetic trap is signaled if an operand is invalid for the operation to be performed. Invalid operations are:

- Any operation on a signaling NaN.
- Addition of unlike-signed infinities or subtraction of like-signed infinities, such as (+\text{infinity} + -\text{infinity}) or (+\text{infinity} - +\text{infinity}).
- Multiplication of 0\times\text{infinity}.
- Division of 0/0 or infinity/infinity.
- Conversion of an infinity or NaN to an integer.
- CMPLTLE or CMPLT LT when either operand is a NaN.

An implementation may also choose to signal an invalid operation when it encounters an operand that is non-finite. However, CMPTxy does not trap on plus or minus infinity.
The instruction cannot disable the trap. If the trap occurs, an UNPREDICTABLE value is stored in the result register.

IEEE-compliant system software must also supply an invalid operation indication to the user for SQRT of a negative non-zero number, for x REM 0, and for conversions to integer that take an integer overflow trap. If an implementation does not support the division by zero disable bit (DZED), it may respond to the division of 0/0 by delivering a division by zero trap to the operating system, which IEEE compliant software must change to an invalid operation trap for the user.

4.7.6.3 Division by Zero (DZE) Arithmetic Trap

A division by zero arithmetic trap is taken if the numerator does not cause an invalid operation trap and the denominator is zero.

The instruction cannot disable the trap. If the trap occurs, an UNPREDICTABLE value is stored in the result register.

If an implementation does not support the division by zero disable bit (DZED), it may respond to the division of 0/0 by delivering a division by zero trap to the operating system, which IEEE compliant software must change to an invalid operation trap for the user.

4.7.6.4 Overflow (OVF) Arithmetic Trap

An overflow arithmetic trap is signaled if the rounded result exceeds in magnitude the largest finite number of the destination format.

The instruction cannot disable the trap. If the trap occurs, an UNPREDICTABLE value is stored in the result register.

4.7.6.5 Underflow (UNF) Arithmetic Trap

An underflow occurs if the rounded result is smaller in magnitude than the smallest finite number of the destination format.

If an underflow occurs, a true zero (64 bits of zero) is always stored in the result register, even if the proper IEEE result would have been -0 (underflow below the negative denormal range).

If an underflow occurs and underflow traps are enabled by the instruction, an underflow arithmetic trap is signaled.

4.7.6.6 Inexact Result (INE) Arithmetic Trap

An inexact result occurs if the infinitely precise result differs from the rounded result.

If an inexact result occurs, the normal rounded result is still stored in the result register.

If an inexact result occurs and inexact result traps are enabled by the instruction, an inexact result arithmetic trap is signaled.
4.7.6.7 Integer Overflow (IOV) Arithmetic Trap

In conversions from floating to quadword integer, an integer overflow occurs if the rounded result is outside the range \(-2^{**}63..2^{**}63-1\). In conversions from quadword integer to longword integer, an integer overflow occurs if the result is outside the range \(-2^{**}31..2^{**}31-1\).

If an integer overflow occurs in CVTxQ or CVTQL, the true result truncated to the low-order 64 or 32 bits respectively is stored in the result register.

If an integer overflow occurs and integer overflow traps are enabled by the instruction, an integer overflow arithmetic trap is signaled.

4.7.6.8 Floating-Point Trap Disable Bits

In the case of IEEE software completion trap modes, any of the traps described in Sections 4.7.6.2 through 4.7.6.7 may be disabled by setting the corresponding trap disable bit in the FPCR. The trap disable bits only affect the corresponding IEEE trap modes when the instruction is coupled with the /S (software completion) qualifier. The trap disable bits do not affect any of the VAX trap modes.

If a trap disable bit is set and the corresponding trap condition occurs, the hardware implementation sets the result of the operation to the nontrapping result value as specified in the IEEE standard and Section 4.7.10 and modified by the underflow to zero (UNDZ) bit. If the implementation is unable to calculate the required result, it ignores the trap disable bit and signals a trap as usual. (When an implementation supports both the underflow disable bit and the underflow to zero bit, and both bits are set in the FPCR, the implementation sets the result of an underflow operation to an appropriately signed true zero value.)

Note that a hardware implementation may choose to support any subset of the trap disable bits, including the empty subset.

4.7.7 FPCR Register and Dynamic Rounding Mode

When an IEEE floating-point operate instruction specifies dynamic mode (/D) in its function field (function field bits \(<12:11> = 11\), the rounding mode to be used for the instruction is derived from the FPCR register. The layout of the rounding mode bits and their assignments matches exactly the format used in the 11-bit function field of the floating-point operate instructions. The function field is described in Section 4.7.9.

In addition, the FPCR gives a summary of each exception type for the exception conditions detected by all IEEE floating-point operates thus far, as well as an overall summary bit that indicates whether any of these exception conditions has been detected. The individual exception bits match exactly in purpose and order the exceptions bits found in the exception summary quadword that is pushed for arithmetic traps. However, for each instruction, these exceptions bits are set independent of the trapping mode specified for the instruction. Therefore, even though trapping may be disabled for a certain exceptional condition, the fact that the exceptional condition was encountered by an instruction will still be recorded in the FPCR.
Floating-point operates that belong to the IEEE subset and CVTQL, which belongs to both VAX and IEEE subsets, appropriately set the FPCR exception bits. It is UNPREDICTABLE whether floating-point operates that belong only to the VAX floating-point subset set the FPCR exception bits.

Alpha floating-point hardware only transitions these exception bits from zero to one. Once set to one, these exception bits are only cleared when software writes zero into these bits by writing a new value into the FPCR.

The five trap disable bits may be subsetted in the hardware implementation. Any unimplemented bits are read as zero and ignored when set; the hardware behaves as if unimplemented bits are zero. In addition:

- If the UNFD bit is not implemented, the hardware may not implement the UNDZ bit.
- If the DZED bit is implemented, division of 0/0 must be treated as an invalid operation instead of a division by zero.

The format of the FPCR is shown in Figure 4–1 and described in Table 4–8.

Figure 4–1: Floating-Point Control Register (FPCR) Format

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>60</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>U</td>
<td>M</td>
<td>I</td>
</tr>
<tr>
<td>O</td>
<td>V</td>
<td>F</td>
<td>D</td>
</tr>
<tr>
<td>Z</td>
<td>E</td>
<td>I</td>
<td>N</td>
</tr>
<tr>
<td>F</td>
<td>O</td>
<td>V</td>
<td>D</td>
</tr>
<tr>
<td>R</td>
<td>A</td>
<td>Z</td>
<td>I</td>
</tr>
</tbody>
</table>

Table 4–8: Floating-Point Control Register (FPCR) Bit Descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description (Meaning When Set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>Summary Bit (SUM). Records bitwise OR of FPCR exception bits. Equal to FPCR&lt;57</td>
</tr>
<tr>
<td>62</td>
<td>Inexact Disable (INED). Suppress INE trap and place correct IEEE nontrapping result in the destination register.</td>
</tr>
<tr>
<td>61</td>
<td>Underflow Disable (UNFD). Suppress UNF trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping result. The correct result value is determined according to the value of the UNDZ bit.</td>
</tr>
<tr>
<td>60</td>
<td>Underflow to Zero (UNDZ). When set together with UNFD, on underflow the hardware places a true zero (64 bits of zero) in the destination register rather than the denormal number specified by the IEEE standard.</td>
</tr>
</tbody>
</table>
Table 4–8 (Cont.): Floating-Point Control Register (FPCR) Bit Descriptions

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description (Meaning When Set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>59–58</td>
<td>Dynamic Rounding Mode (DYN). Indicates the rounding mode to be used by an IEEE floating-point operate instruction when the instruction's function field specifies dynamic mode (/D). Assignments are:</td>
</tr>
<tr>
<td></td>
<td><strong>DYN</strong></td>
</tr>
<tr>
<td>00</td>
<td>Chopped rounding mode</td>
</tr>
<tr>
<td>01</td>
<td>Minus infinity</td>
</tr>
<tr>
<td>10</td>
<td>Normal rounding</td>
</tr>
<tr>
<td>11</td>
<td>Plus infinity</td>
</tr>
<tr>
<td>57</td>
<td>Integer Overflow (IOV). An integer arithmetic operation or a conversion from floating to integer overflowed the destination precision.</td>
</tr>
<tr>
<td>56</td>
<td>Inexact Result (INE). A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.</td>
</tr>
<tr>
<td>55</td>
<td>Underflow (UNF). A floating arithmetic or conversion operation underflowed the destination exponent.</td>
</tr>
<tr>
<td>54</td>
<td>Overflow (OVF). A floating arithmetic or conversion operation overflowed the destination exponent.</td>
</tr>
<tr>
<td>53</td>
<td>Division by Zero (DZE). An attempt was made to perform a floating divide operation with a divisor of zero.</td>
</tr>
<tr>
<td>52</td>
<td>Invalid Operation (INV). An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.</td>
</tr>
<tr>
<td>51</td>
<td>Overflow Disable (OVFD). Suppress OVF trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping result.</td>
</tr>
<tr>
<td>50</td>
<td>Division by Zero Disable (DZED). Suppress DZE trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping result.</td>
</tr>
<tr>
<td>49</td>
<td>Invalid Operation Disable (INVD). Suppress INV trap and place correct IEEE nontrapping result in the destination register if the implementation is capable of producing correct IEEE nontrapping result.</td>
</tr>
<tr>
<td>48-0</td>
<td>Reserved. Read As Zero; Ignored when written.</td>
</tr>
</tbody>
</table>

FPCR is read from and written to the floating-point registers by the MT_FPCR and MF_FPCR instructions respectively, which are described in Section 4.7.7.1.

FPCR and the instructions to access it are required for an implementation that supports floating-point (see Section 4.7.7). On implementations that do not support
floating-point, the instructions that access FPCR (MF_FPCR and MT_FPCR) take an Illegal Instruction Trap.

Software Note:
Support for FPCR is required on a system that supports the OpenVMS Alpha operating system even if that system does not support floating-point.

4.7.7.1 Accessing the FPCR

Because Alpha floating-point hardware can overlap the execution of a number of floating-point instructions, accessing the FPCR must be synchronized with other floating-point instructions. An EXCB instruction must be issued both prior to and after accessing the FPCR to ensure that the FPCR access is synchronized with the execution of previous and subsequent floating-point instructions; otherwise synchronization is not ensured.

Issuing an EXCB followed by an MT_FPCR followed by another EXCB ensures that only floating-point instructions issued after the second EXCB are affected by and affect the new value of the FPCR. Issuing an EXCB followed by an MF_FPCR followed by another EXCB ensures that the value read from the FPCR only records the exception information for floating-point instructions issued prior to the first EXCB.

Consider the following example:

```
ADDT/D
EXCB ;1
MT_FPCR F1,F1,F1
EXCB ;2
SUBT/D
```

Without the first EXCB, it is possible in an implementation for the ADDT/D to execute in parallel with the MT_FPCR. Thus, it would be UNPREDICTABLE whether the ADDT/D was affected by the new rounding mode set by the MT_FPCR and whether fields cleared by the MT_FPCR in the exception summary were subsequently set by the ADDT/D.

Without the second EXCB, it is possible in an implementation for the MT_FPCR to execute in parallel with the SUBT/D. Thus, it would be UNPREDICTABLE whether the SUBT/D was affected by the new rounding mode set by the MT_FPCR and whether fields cleared by the MT_FPCR in the exception summary field of FPCR were previously set by the SUBT/D.

Specifically, code should issue an EXCB before and after it accesses the FPCR if that code needs to see valid values in FPCR bits <63> and <57:52>. An EXCB should be issued before attempting to write the FPCR if the code expects changes to bits <59:52> not to have dependencies with prior instructions. An EXCB should be issued after attempting to write the FPCR if the code expects subsequent instructions to have dependencies with changes to bits <59:52>.
4.7.7.2 Default Values of the FPCR

Processor initialization leaves the value of FPCR UNPREDICTABLE.

Software Note:

Digital software should initialize FPCR<\textsc{Dyn}> = 10 during program activation. Using this default, a program can be coded to use only dynamic rounding without the need to explicitly set the rounding mode to normal rounding in its start-up code.

Program activation normally clears all other fields in the FPCR. However, this behavior may depend on the operating system.

4.7.7.3 Saving and Restoring the FPCR

The FPCR must be saved and restored across context switches so that the FPCR value of one process does not affect the rounding behavior and exception summary of another process.

The dynamic rounding mode put into effect by the programmer (or initialized by image activation) is valid for the entirety of the program and remains in effect until subsequently changed by the programmer or until image run-down occurs.

Software Notes:

The following software notes apply to saving and restoring the FPCR:

1. The IEEE standard precludes saving and restoring the FPCR across subroutine calls.

2. The IEEE standard requires that an implementation provide status flags that are set whenever the corresponding conditions occur and are reset only at the user’s request. The exception bits in the FPCR do not satisfy that requirement, because they can be spuriously set by instructions in a trap shadow that should not have been executed had the trap been taken synchronously.

The IEEE status flags can be provided by software (as software status bits) as follows:

Trap interface software (usually the operating system) keeps a set of software status bits and a mask of the traps that the user wants to receive. Code is generated with the /SUI modifiers. For a particular exception, the software clears the corresponding trap disable bit if either the corresponding software status bit is 0 or if the user wants to receive such traps. If a trap occurs, the software locates the offending instruction in the trap shadow, simulates it and sets any of the software status bits that are appropriate. Then, the software either delivers the trap to the user program or disables further delivery of such traps. The user program must interface to this trap interface software to set or clear any of the software status bits or to enable or disable floating-point traps.
When such a scheme is being used, the trap disable bits should be modified only by the trap interface software. If the disable bits are spuriously cleared, unnecessary traps may occur. If they are spuriously set, the software may fail to set the correct values in the software status bits. Programs should call routines in the trap interface software to set or clear bits in the FPCR.

Digital software may choose to initialize the software status bits and the trap disable bits to all 1's to avoid any initial trapping when an exception condition first occurs. Or, software may choose to initialize those bits to all 0's in order to provide a summary of the exception behavior when the program terminates.

In any event, the exception bits in the FPCR are still useful to programs. A program can clear all of the exception bits in the FPCR, execute a single floating-point instruction, and then examine the status bits to determine which hardware-defined exceptions the instruction encountered. For this operation to work in the presence of various implementation options, the single instruction should be followed by a TRAPB or EXCB instruction, and software completion by the system software should save and restore the FPCR registers without other modifications.

3. Because of the way the LDS and STS instructions manipulate bits <61:59> of floating-point registers, they should not be used to manipulate FPCR values.

4.7.8 Floating-Point Computational Models

There are three models of arithmetic available with the IEEE floating-point subset in the Alpha architecture:

- IEEE compliant arithmetic
- IEEE compliant arithmetic without inexact exception
- High-performance IEEE-format arithmetic

**IEEE Compliant Arithmetic**

This model provides floating-point arithmetic that fully complies with the IEEE standard. It provides all of the exception status flags that are in the standard and allows the user to specify which exceptional conditions should trap and which should proceed without trapping.

This model is implemented in a program by using IEEE floating-point operates with the /SUI modifiers, with the help of the trap interface software described in Software Note 2 in Section 4.7.7.3. It provides acceptable performance on implementations of the Alpha architecture that implement the inexact disable (INED) bit in the FPCR, as long as such programs do not turn on traps for the inexact condition. Performance under this model may be slow if the INED bit is not implemented.

**IEEE Compliant Arithmetic Without Inexact Exception**

This is similar to the previous model, but it does not provide the inexact exception status bit, nor does it allow a program to request traps when an inexact operation occurs.
This model is implemented in a program by using IEEE floating-point operates with the /SU modifiers, with the help of the trap interface software.

High-Performance IEEE-Format Arithmetic
This model provides arithmetic operations on IEEE format numbers, but does not allow operations on or generation of non-finite numbers. Any attempt to operate on a non-finite number may cause an unrecoverable trap, and any operation except underflow that would generate a non-finite number (according to the IEEE standard) may also cause an unrecoverable trap. Underflow results are set to zero. There are no reliable IEEE exception status flags available.

This model is implemented in a program by using IEEE floating-point operates without the /SU or /SUI modifiers. It is the fastest of the three models.

4.7.9 Floating-Point Instruction Function Field Format
The function code for IEEE and VAX floating-point instructions, bits <15..5>, contain the function field. That field is shown in Figure 4–2 and described for IEEE floating-point in Table 4–9 and for VAX floating-point in Table 4–10. Function codes for the independent floating-point instructions, those with opcode 1716, do not correspond to the function fields below.

The function field contains subfields that specify the trapping and rounding modes that are enabled for the instruction, the source datatype, and the instruction class.

Figure 4–2: Floating-Point Instruction Function Field
<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Meaning¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>15–13</td>
<td>TRP</td>
<td>Trapping modes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000 Imprecise (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001 Underflow enable (/U) — floating-point output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer overflow enable (/V) — integer output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>011 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101 /SU — floating-point output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/SV — integer output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111 /SUI — floating-point output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/SVI — integer output</td>
</tr>
<tr>
<td>12–11</td>
<td>RND</td>
<td>Rounding modes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 Chopped (/C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 Minus infinity (/M)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 Normal (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 Dynamic (/D)</td>
</tr>
<tr>
<td>10–9</td>
<td>SRC</td>
<td>Source datatype:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 S_floating</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 T_floating</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 Q_fixed</td>
</tr>
</tbody>
</table>

¹Encodings for the instructions CVTST and CVTST/S are exceptions to this table; use the encodings in Appendix C.
Table 4–9 (Cont.): IEEE Floating-Point Function Field Bit Summary

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-5</td>
<td>FNC</td>
<td>Instruction class:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000 ADDx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001 SUBx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010 MULx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011 DIVx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100 CMPxUN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101 CMPxEQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110 CMPxLT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111 CMPxLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100 CVTxS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1101 Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1110 CVTxT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111 CVTxQ</td>
</tr>
</tbody>
</table>

1Encodings for the instructions CVTST and CVTST/S are exceptions to this table; use the encodings in Appendix C.

Table 4–10: VAX Floating-Point Function Field Bit Summary

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>15–13</td>
<td>TRP</td>
<td>Trapping modes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000 Imprecise (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100 Software completion enable (/S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101 /SU — floating-point output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/SV — integer output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111 Unsupported</td>
</tr>
<tr>
<td>12–11</td>
<td>RND</td>
<td>Rounding modes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 Chopped (/C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 Normal (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 Unsupported</td>
</tr>
</tbody>
</table>
Table 4–10 (Cont.): VAX Floating-Point Function Field Bit Summary

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>10–9</td>
<td>SRC</td>
<td>Source datatype:</td>
</tr>
<tr>
<td>00</td>
<td>F_floating</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>D_floating</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>G_floating</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Q_fixed</td>
<td></td>
</tr>
<tr>
<td>8–5</td>
<td>FNC</td>
<td>Instruction class:</td>
</tr>
<tr>
<td>0000</td>
<td>ADDx</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>SUBx</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>MULx</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>DIVx</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>CMPxUN</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>CMPxEQ</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>CMPxLT</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>CMPxLE</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>CVTxF</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>CVTxD</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>CVTxG</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>CVTxQ</td>
<td></td>
</tr>
</tbody>
</table>

4.7.10 IEEE Standard


This standard leaves certain operations as implementation dependent. The remainder of this section specifies the behavior of the Alpha architecture in these situations. Note that this behavior may be supplied by either hardware (if the invalid operation disable, or INVD, bit is implemented) or by software. See Sections 4.7.6.8, 4.7.7, and 4.7.7.3.

4.7.10.1 Conversion of NaN and Infinity Values

Conversion of a NaN or an Infinity value to an integer gives a result of zero.

Conversion of a NaN value from S_floating to T_floating gives a result identical to the input, except that the most significant fraction bit (bit 51) is set to indicate a quiet NaN.
Conversion of a NaN value from T_floating to S_floating gives a result identical to the input, except that the most significant fraction bit (bit 51) is set to indicate a quiet NaN, and bits <28:0> are cleared to zero.

4.7.10.2 Copying NaN Values
Copy a NaN value without changing its precision does not cause an invalid operation exception.

4.7.10.3 Generating NaN Values
When an operation is required to produce a NaN and none of its inputs are NaN values, the result of the operation is the quiet NaN value that has the sign bit set to one, all exponent bits set to one (to indicate a NaN), the most significant fraction bit set to one (to indicate that the NaN is quiet), and all other fraction bits cleared to zero. This value is referred to as the “canonical quiet NaN.”

4.7.10.4 Propagating NaN Values
When an operation is required to produce a NaN and one or both of its inputs are NaN values, the IEEE standard requires that quiet NaN values be propagated when possible. With the Alpha architecture, the result of such an operation is a NaN generated according to the first of the following rules that is applicable:

1. If the operand in the Fb register of the operation is a quiet NaN, that value is used as the result.
2. If the operand in the Fb register of the operation is a signaling NaN, the result is the quiet NaN formed from the Fb value by setting the most significant fraction bit (bit 51) to a one bit.
3. If the operation uses its Fa operand and the value in the Fa register is a quiet NaN, that value is used as the result.
4. If the operation uses its Fa operand and the value in the Fa register is a signaling NaN, the result is the quiet NaN formed from the Fa value by setting the most significant fraction bit (bit 51) to a one bit.
5. The result is the canonical quiet NaN.
4.8 Memory Format Floating-Point Instructions

The instructions in this section move data between the floating-point registers and memory. They use the Memory instruction format. They do not interpret the bits moved in any way; specifically, they do not trap on non-finite values.

The instructions are summarized in Table 4-11.

Table 4–11: Memory Format Floating-Point Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDF</td>
<td>Load F_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>LDG</td>
<td>Load G_floating (Load D_floating)</td>
<td>VAX</td>
</tr>
<tr>
<td>LDS</td>
<td>Load S_floating (Load Longword Integer)</td>
<td>Both</td>
</tr>
<tr>
<td>LDT</td>
<td>Load T_floating (Load Quadword Integer)</td>
<td>Both</td>
</tr>
<tr>
<td>STF</td>
<td>Store F_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>STG</td>
<td>Store G_floating (Store D_floating)</td>
<td>VAX</td>
</tr>
<tr>
<td>STS</td>
<td>Store S_floating (Store Longword Integer)</td>
<td>Both</td>
</tr>
<tr>
<td>STT</td>
<td>Store T_floating (Store Quadword Integer)</td>
<td>Both</td>
</tr>
</tbody>
</table>
4.8.1 Load F_floating

Format:

\[
\text{LDF } \quad \text{Fa.wf,disp.ab(Rb.ab)} \quad \text{!Memory format}
\]

Operation:

\[
\text{va } \leftarrow \{\text{Rbv + SEXT(disp)}\}
\]

\[
\text{CASE}
\]

\[
\text{big_endian_data: va'} \leftarrow \text{va XOR 1002}
\]

\[
\text{little_endian_data: va'} \leftarrow \text{va}
\]

\[
\text{ENDCASE}
\]

\[
\text{Fa } \leftarrow \begin{array}{c}
\text{(va'}<15> \mid \mid \text{MAP_F((va'}<14:7>)} \mid \\
\text{(va'}<6:0> \mid \mid \text{(va'}<31:16} \mid \mid 0<28:0>
\end{array}
\]

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDF Load F_floating

Qualifiers:

None

Description:

LDF fetches an F_floating datum from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated.

The MAP_F function causes the 8-bit memory-format exponent to be expanded to an 11-bit register-format exponent according to Table 2–1.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. For a big-endian longword access, va<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va'). The source operand is fetched from memory and the bytes are reordered to conform to the F_floating register format. The result is then zero-extended in the low-order longword and written to register Fa.
4.8.2 Load G_floating

Format:

\[
\text{LDG} \quad \text{Fa.wg,disp.ab(Rb.ab)} \quad \text{!Memory format}
\]

Operation:

\[
\begin{align*}
\text{va} & \leftarrow \{Rbv + \text{SEXT(disp)}\} \\
\text{Fa} & \leftarrow (\text{va})_{15:0} | (\text{va})_{31:16} | (\text{va})_{47:32} | (\text{va})_{63:48}
\end{align*}
\]

Exceptions:

- Access Violation
- Fault on Read
- Alignment
- Translation Not Valid

Instruction mnemonics:

LDG Load G_floating (Load D_floating)

Qualifiers:

None

Description:

LDG fetches a G_floating (or D_floating) datum from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory, the bytes are reordered to conform to the G_floating register format (also conforming to the D_floating register format), and the result is then written to register Fa.
4.8.3 Load S_floating

Format:

LDS Fa.ws,disp.ab(Rb.ab) !Memory format

Operation:

va ← \{Rbv + SEXT(disp)}
CASE
  big_endian_data: va' ← va XOR 1002
  little_endian_data: va' ← va
ENDCASE
Fa ← (va')<31> || MAP_S((va')<30:23>) || (va')<22:0> || 0<28:0>

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDS Load S_floating (Load Longword Integer)

Qualifiers:

None

Description:

LDS fetches a longword (integer or S_floating) from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated. The MAP_S function causes the 8-bit memory-format exponent to be expanded to an 11-bit register-format exponent according to Table 2-2.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. For a big-endian longword access, va<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va'). The source operand is fetched from memory, is zero-extended in the low-order longword, and then written to register Fa. Longword integers in floating registers are stored in bits <63:62,58:29>, with bits <61:59> ignored and zeros in bits <28:0>.
4.8.4 Load T_floating

Format:

\[ \text{LDT} \quad \text{Fa.wt,disp.ab(Rb.ab)} \quad \text{!Memory format} \]

Operation:

\[ \text{va} \leftarrow \{\text{Rbv + SEXT(disp)}\} \]
\[ \text{Fa} \leftarrow (\text{va})^{<63:0}> \]

Exceptions:

Access Violation
Fault on Read
Alignment
Translation Not Valid

Instruction mnemonics:

LDT Load T_floating (Load Quadword Integer)

Qualifiers:

None

Description:

LDT fetches a quadword (integer or T_floating) from memory and writes it to register Fa. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from memory and written to register Fa.
4.8.5 Store F_floating

Format:

```
STF Fa.rf,disp.ab(Rb.ab) !Memory format
```

Operation:

```
va ← {Rbv + SEXT(disp)}
CASE
  big_endian_data:  va’ ← va XOR 1002
  little_endian_data: va’ ← va
ENDCASE
(va’)<31:0> ← Fav<44:29> | | Fav<63:62>| | Fav<58:45>
```

Exceptions:

- Access Violation
- Fault on Write
- Alignment
- Translation Not Valid

Instruction mnemonics:

```
STF Store F_floating
```

Qualifiers:

None

Description:

STF stores an F_floating datum from Fa to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. For a big-endian longword access, va<2> (bit 2 of the virtual address) is inverted, and any memory management fault is reported for va (not va’). The bits of the source operand are fetched from register Fa, the bits are reordered to conform to F_floating memory format, and the result is then written to memory. Bits <61:59> and <28:0> of Fa are ignored. No checking is done.
4.8.6 Store G_floating

Format:

```
STG Fa,rg,disp.ab(Rb.ab) !Memory format
```

Operation:

```
va ← \{Rbv + SEXT(disp)}
(va)<63:0> ← Fav<15:0> || Fav<31:16> ||
Fav<47:32> || Fav<63:48>
```

Exceptions:

- Access Violation
- Fault on Write
- Alignment
- Translation Not Valid

Instruction mnemonics:

```
STG Store G_floating (Store D_floating)
```

Qualifiers:

None

Description:

STG stores a G_floating (or D_floating) datum from Fa to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from register Fa, the bytes are reordered to conform to the G_floating memory format (also conforming to the D_floating memory format), and the result is then written to memory.
### 4.8.7 Store \( S_{\text{floating}} \)

**Format:**

\[
\text{STS} \quad \text{Fa.rs,disp.ab(Rb.ab)} \quad \text{Memory format}
\]

**Operation:**

\[
\begin{align*}
\text{va} & \leftarrow \{\text{Rbv} + \text{SEXT(disp)}\} \\
\text{CASE} & \\
\text{big_endian_data:} & \quad \text{va’} \leftarrow \text{va XOR 100} \\
\text{little_endian_data:} & \quad \text{va’} \leftarrow \text{va} \\
\text{ENDCASE} \\
(\text{va’})<31:0> & \leftarrow \text{Fav}<63:62>||\text{Fav}<58:29>
\end{align*}
\]

**Exceptions:**

- Access Violation
- Fault on Write
- Alignment
- Translation Not Valid

**Instruction mnemonics:**

- **STS**  
  Store \( S_{\text{floating}} \) (Store Longword Integer)

**Qualifiers:**

None

**Description:**

STS stores a longword (integer or \( S_{\text{floating}} \)) datum from \( \text{Fa} \) to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register \( \text{Rb} \) to the sign-extended 16-bit displacement. For a big-endian longword access, \( \text{va}<2> \) (bit 2 of the virtual address) is inverted, and any memory management fault is reported for \( \text{va} \) (not \( \text{va’} \)). The bits of the source operand are fetched from register \( \text{Fa} \), the bits are reordered to conform to \( S_{\text{floating}} \) memory format, and the result is then written to memory. Bits \( <61:59> \) and \( <28:0> \) of \( \text{Fa} \) are ignored. No checking is done.
4.8.8 Store T_floating

Format:

```
STT Fa.rt,disp.ab(Rb.ab) !Memory format
```

Operation:

```
va ← {Rbv + SEXT(disp)}
(va)<63:0> ← Fav<63:0>
```

Exceptions:

- Access Violation
- Fault on Write
- Alignment
- Translation Not Valid

Instruction mnemonics:

```
STT Store T_floating (Store Quadword Integer)
```

Qualifiers:

None

Description:

STT stores a quadword (integer or T_floating) datum from Fa to memory. If the data is not naturally aligned, an alignment exception is generated.

The virtual address is computed by adding register Rb to the sign-extended 16-bit displacement. The source operand is fetched from register Fa and written to memory.
4.9 Branch Format Floating-Point Instructions

Alpha provides six floating conditional branch instructions. These branch-format instructions test the value of a floating-point register and conditionally change the PC.

They do not interpret the bits tested in any way; specifically, they do not trap on non-finite values.

The test is based on the sign bit and whether the rest of the register is all zero bits. All 64 bits of the register are tested. The test is independent of the format of the operand in the register. Both plus and minus zero are equal to zero. A non-zero value with a sign of zero is greater than zero. A non-zero value with a sign of one is less than zero. No reserved operand or non-finite checking is done.

The floating-point branch operations are summarized in Table 4–12.

Table 4–12: Floating-Point Branch Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBEQ</td>
<td>Floating Branch Equal</td>
<td>Both</td>
</tr>
<tr>
<td>FBGE</td>
<td>Floating Branch Greater Than or Equal</td>
<td>Both</td>
</tr>
<tr>
<td>FBGT</td>
<td>Floating Branch Greater Than</td>
<td>Both</td>
</tr>
<tr>
<td>FBLE</td>
<td>Floating Branch Less Than or Equal</td>
<td>Both</td>
</tr>
<tr>
<td>FBLT</td>
<td>Floating Branch Less Than</td>
<td>Both</td>
</tr>
<tr>
<td>FBNE</td>
<td>Floating Branch Not Equal</td>
<td>Both</td>
</tr>
</tbody>
</table>
4.9.1 Conditional Branch

Format:

FBxx Fa.rq,disp.al !Branch format

Operation:

{update PC}
va ← PC + {4*SEXT(disp)}
IF TEST(Fav, Condition_based_on_Opcode) THEN
PC ← va

Exceptions:

None

Instruction mnemonics:

FBEQ Floating Branch Equal
FBGE Floating Branch Greater Than or Equal
FBGT Floating Branch Greater Than
FBLE Floating Branch Less Than or Equal
FBLT Floating Branch Less Than
FBNE Floating Branch Not Equal

Qualifiers:

None

Description:

Register Fa is tested. If the specified relationship is true, the PC is loaded with the target virtual address; otherwise, execution continues with the next sequential instruction.

The displacement is treated as a signed longword offset. This means it is shifted left two bits (to address a longword boundary), sign-extended to 64 bits, and added to the updated PC to form the target virtual address.

The conditional branch instructions are PC-relative only. The 21-bit signed displacement gives a forward/backward branch distance of +/- 1M instructions.
Notes:

- To branch properly on non-finite operands, compare to F31, then branch on the result of the compare.

- The largest negative integer (8000 0000 0000 0000_{16}) is the same bit pattern as floating minus zero, so it is treated as equal to zero by the branch instructions. To branch properly on the largest negative integer, convert it to floating or move it to an integer register and do an integer branch.
4.10 Floating-Point Operate Format Instructions

The floating-point bit-operate instructions perform copy and integer convert operations on 64-bit register values. The bit-operate instructions do not interpret the bits moved in any way; specifically, they do not trap on non-finite values.

The floating-point arithmetic-operate instructions perform add, subtract, multiply, divide, compare, and floating convert operations on 64-bit register values in one of the four specified floating formats.

Each instruction specifies the source and destination formats of the values, as well as the rounding mode and trapping mode to be used. These instructions use the Floating-point Operate format.

The floating-point operate instructions are summarized in Table 4–13.

Table 4–13: Floating-Point Operate Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPYS</td>
<td>Copy Sign</td>
<td>Both</td>
</tr>
<tr>
<td>CPYSE</td>
<td>Copy Sign and Exponent</td>
<td>Both</td>
</tr>
<tr>
<td>CPYSN</td>
<td>Copy Sign Negate</td>
<td>Both</td>
</tr>
<tr>
<td>CVTLQ</td>
<td>Convert Longword to Quadword</td>
<td>Both</td>
</tr>
<tr>
<td>CVTQL</td>
<td>Convert Quadword to Longword</td>
<td>Both</td>
</tr>
<tr>
<td>FCMOVxx</td>
<td>Floating Conditional Move</td>
<td>Both</td>
</tr>
<tr>
<td>MF_FPCR</td>
<td>Move from Floating-point Control Register</td>
<td>Both</td>
</tr>
<tr>
<td>MT_FPCR</td>
<td>Move to Floating-point Control Register</td>
<td>Both</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation</td>
<td>Subset</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ADDF</td>
<td>Add $F_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>ADDG</td>
<td>Add $G_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>ADDS</td>
<td>Add $S_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>ADDT</td>
<td>Add $T_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>CMPGxx</td>
<td>Compare $G_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>CMPTxx</td>
<td>Compare $T_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>CVTDG</td>
<td>Convert $D_{\text{floating}}$ to $G_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>CVTGD</td>
<td>Convert $G_{\text{floating}}$ to $D_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>CVTGF</td>
<td>Convert $G_{\text{floating}}$ to $F_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>CVTQG</td>
<td>Convert $G_{\text{floating}}$ to Quadword</td>
<td>VAX</td>
</tr>
<tr>
<td>CVTQF</td>
<td>Convert Quadword to $F_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>CVTQG</td>
<td>Convert Quadword to $G_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>CVTQS</td>
<td>Convert Quadword to $S_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>CVTQT</td>
<td>Convert Quadword to $T_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>CVTST</td>
<td>Convert $S_{\text{floating}}$ to $T_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>CVTTQ</td>
<td>Convert $T_{\text{floating}}$ to Quadword</td>
<td>IEEE</td>
</tr>
<tr>
<td>CVTTS</td>
<td>Convert $T_{\text{floating}}$ to $S_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>DIVF</td>
<td>Divide $F_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>DIVG</td>
<td>Divide $G_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>DIVS</td>
<td>Divide $S_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>DIVT</td>
<td>Divide $T_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>FTOIS</td>
<td>Floating-point to integer register move, $S_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>FTOIT</td>
<td>Floating-point to integer register move, $T_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>ITOFF</td>
<td>Integer to floating-point register move, $F_{\text{floating}}$</td>
<td>VAX</td>
</tr>
<tr>
<td>ITOFS</td>
<td>Integer to floating-point register move, $S_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>ITOFT</td>
<td>Integer to floating-point register move, $T_{\text{floating}}$</td>
<td>IEEE</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation</td>
<td>Subset</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------</td>
<td>--------</td>
</tr>
<tr>
<td>MULF</td>
<td>Multiply F_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>MULG</td>
<td>Multiply G_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>MULS</td>
<td>Multiply S_floating</td>
<td>IEEE</td>
</tr>
<tr>
<td>MULT</td>
<td>Multiply T_floating</td>
<td>IEEE</td>
</tr>
<tr>
<td>SQRTF</td>
<td>Square root F_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>SQRTG</td>
<td>Square root G_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>SQRTS</td>
<td>Square root S_floating</td>
<td>IEEE</td>
</tr>
<tr>
<td>SQRTT</td>
<td>Square root T_floating</td>
<td>IEEE</td>
</tr>
<tr>
<td>SUBF</td>
<td>Subtract F_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>SUBG</td>
<td>Subtract G_floating</td>
<td>VAX</td>
</tr>
<tr>
<td>SUBS</td>
<td>Subtract S_floating</td>
<td>IEEE</td>
</tr>
<tr>
<td>SUBT</td>
<td>Subtract T_floating</td>
<td>IEEE</td>
</tr>
</tbody>
</table>
4.10.1 Copy Sign

Format:

CPYSy Fa.rq,Fb.rq,Fc.wq !Floating-point Operate format

Operation:

CASE
  CPYS: Fc ← Fav<63> || Fbv<62:0>
  CPYSN: Fc ← NOT(Fav<63>) || Fbv<62:0>
  CPYSE: Fc ← Fav<63:52> || Fbv<51:0>
ENDCASE

Exceptions:

None

Instruction mnemonics:

CPYS Copy Sign
CPYSE Copy Sign and Exponent
CPYSN Copy Sign Negate

Qualifiers:

None

Description:

For CPYS and CPYSN, the sign bit of Fa is fetched (and complemented in the case of CPYSN) and concatenated with the exponent and fraction bits from Fb; the result is stored in Fc.

For CPYSE, the sign and exponent bits from Fa are fetched and concatenated with the fraction bits from Fb; the result is stored in Fc.

No checking of the operands is performed.

Notes:

- Register moves can be performed using CPYS Fx,Fx,Fy. Floating-point absolute value can be done using CPYS F31,Fx,Fy. Floating-point negation can be done using CPYSN Fx,Fx,Fy. Floating values can be scaled to a known range by using CPYSE.
4.10.2 Convert Integer to Integer

Format:

\[ \text{CVT}_{xy} \ Fb.rq,Fc.wx \] !Floating-point Operate format

Operation:

CASE
\[ \begin{align*}
\text{CVTQL: } Fc & \leftarrow Fb<31:30> \ || 0<2:0> \ || Fb<29:0> \ || 0<28:0> \\
\text{CVTLQ: } Fc & \leftarrow \text{SEXT}(Fb<63:62> \ || Fb<58:29>) \\
\end{align*} \]
ENDCASE

Exceptions:

Integer Overflow, CVTQL only

Instruction mnemonics:

- CVTLQ: Convert Longword to Quadword
- CVTQL: Convert Quadword to Longword

Qualifiers:

- Trapping: Software (/S) (CVTQL only)
- Integer Overflow Enable (/V) (CVTQL only)

Description:

The two's-complement operand in register Fb is converted to a two's-complement result and written to register Fc. Register Fa must be F31.

The conversion from quadword to longword is a repositioning of the low 32 bits of the operand, with zero fill and optional integer overflow checking. Integer overflow occurs if Fb is outside the range \(-2^{31..2^{31}} - 1\). If integer overflow occurs, the truncated result is stored in Fc, and an arithmetic trap is taken if enabled.

The conversion from longword to quadword is a repositioning of 32 bits of the operand, with sign extension.
4.10.3 Floating-Point Conditional Move

Format:

FCMOVxx Fa.rq,Fb.rq,Fc.wq !Floating-point Operate format

Operation:

IF TEST(Fav, Condition_based_on_Opcode) THEN
    Fc ← Fbv

Exceptions:

None

Instruction mnemonics:

FCMOVEQ    FCMOVE if Register Equal to Zero
FCMOVGE    FCMOVE if Register Greater Than or Equal to Zero
FCMOVG7T   FCMOVE if Register Greater Than Zero
FCMOVLE    FCMOVE if Register Less Than or Equal to Zero
FCMOVLT    FCMOVE if Register Less Than Zero
FCMOVNE    FCMOVE if Register Not Equal to Zero

Qualifiers:

None

Description:

Register Fa is tested. If the specified relationship is true, register Fb is written to register Fc; otherwise, the move is suppressed and register Fc is unchanged. The test is based on the sign bit and whether the rest of the register is all zero bits, as described for floating branches in Section 4.9.
Notes:
Except that it is likely in many implementations to be substantially faster, the
instruction:

FCMOVxx Fa,Fb,Fc

is exactly equivalent to:

FByy Fa,label ; yy = NOT xx
CPYS Fb,Fb,Fc

label: ...

For example, a branchless sequence for:

F1=MAX(F1,F2)

is:

CMPxLT F1,F2,F3 ! F3=one if F1<F2; x=F/G/S/T
FCMOVNE F3,F2,F1 ! Move F2 to F1 if F1<F2
4.10.4 Move from/to Floating-Point Control Register

**Format:**

\[ \text{Mx}_\text{FPCR} \quad \text{F}a.rq,\text{F}a.rq,\text{F}a.wq \]

!Floating-point Operate format

**Operation:**

\[
\text{CASE} \\
\quad \text{MT}_\text{FPCR}: \quad \text{FPCR} \leftarrow \text{Fav} \\
\quad \text{MF}_\text{FPCR}: \quad \text{Fa} \leftarrow \text{FPCR} \\
\text{ENDCASE}
\]

**Exceptions:**

None

**Instruction mnemonics:**

- \text{MF}_\text{FPCR} \quad \text{Move from Floating-point Control Register}
- \text{MT}_\text{FPCR} \quad \text{Move to Floating-point Control Register}

**Qualifiers:**

None

**Description:**

The Floating-point Control Register (FPCR) is read from (\text{MF}_\text{FPCR}) or written to (\text{MT}_\text{FPCR}), a floating-point register. The floating-point register to be used is specified by the \text{Fa}, \text{Fb}, and \text{Fc} fields all pointing to the same floating-point register. If the \text{Fa}, \text{Fb}, and \text{Fc} fields do not all point to the same floating-point register, then it is UNPREDICTABLE which register is used. If the \text{Fa}, \text{Fb}, and \text{Fc} fields do not all point to the same floating-point register, the resulting values in the \text{Fc} register and in \text{FPCR} are UNPREDICTABLE.

If the \text{Fc} field is \text{F}31 in the case of \text{MT}_\text{FPCR}, the resulting value in \text{FPCR} is UNPREDICTABLE.

The use of these instructions and the FPCR are described in Section 4.7.7.
4.10.5 VAX Floating Add

**Format:**

\[
\text{ADDx Fa.rx,Fb.rx,Fc.wx} \quad \text{!Floating-point Operate format}
\]

**Operation:**

\[
\text{Fc} \leftarrow \text{Fav + Fbv}
\]

**Exceptions:**

- Invalid Operation
- Overflow
- Underflow

**Instruction mnemonics:**

- ADDF Add F_float
- ADDG Add G_float

**Qualifiers:**

- Rounding: Chopped (/C)
- Trapping: Software (/S)
  - Underflow Enable (/U)

**Description:**

Register Fa is added to register Fb, and the sum is written to register Fc. The sum is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs. See Section 4.7.6 for details of the stored result on overflow or underflow.
4.10.6 IEEE Floating Add

Format:

\[ \text{ADD}x \quad \text{Fa}_{.rx}, \text{Fb}_{.rx}, \text{Fc}_{.wx} \quad \text{!Floating-point Operate format} \]

Operation:

\[ \text{Fc} \leftarrow \text{Fav} + \text{Fbv} \]

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

ADDs Add S\_floating
ADDT Add T\_floating

Qualifiers:

Rounding: Dynamic (/D)
Minus infinity (/M)
Chopped (/C)

Trapping: Software (/S)
Underflow Enable (/U)
Inexact Enable (/I)

Description:

Register Fa is added to register Fb, and the sum is written to register Fc. The sum is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.
4.10.7 VAX Floating Compare

Format:

```
CMPGyy Fa.rg,Fb.rg,Fc.wq
```

Operation:

```
IF Fav SIGNED_RELATION Fbv THEN
   Fc ← 4000 0000 0000 000016
ELSE
   Fc ← 0000 0000 0000 000016
```

Exceptions:

Invalid Operation

Instruction mnemonics:

- CMPGEQ Compare G_floating Equal
- CMPGLE Compare G_floating Less Than or Equal
- CMPGLT Compare G_floating Less Than

Qualifiers:

Trapping: Software (/S)

Description:

The two operands in Fa and Fb are compared. If the relationship specified by the qualifier is true, a non-zero floating value (0.5) is written to register Fc; otherwise, a true zero is written to Fc.

Comparisons are exact and never overflow or underflow. Three mutually exclusive relations are possible: less than, equal, and greater than.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

Notes:

- Compare Less Than A,B is the same as Compare Greater Than B,A; Compare Less Than or Equal A,B is the same as Compare Greater Than or Equal B,A. Therefore, only the less-than operations are included.
4.10.8 IEEE Floating Compare

Format:

```
CMPTyy Fa.rx,Fb.rx,Fc.wq !Floating-point Operate format
```

Operation:

```
IF Fav SIGNED_RELATION Fbv THEN
Fc ← 4000 0000 0000 000016
ELSE
Fc ← 0000 0000 0000 000016
```

Exceptions:

Invalid Operation

Instruction mnemonics:

```
CMPTEQ Compare T_floating Equal
CMPTLE Compare T_floating Less Than or Equal
CMPTLT Compare T_floating Less Than
CMPTUN Compare T_floating Unordered
```

Qualifiers:

```
Trapping: Software (/SU)
```

Description:

The two operands in Fa and Fb are compared. If the relationship specified by the qualifier is true, a non-zero floating value (2.0) is written to register Fc; otherwise, a true zero is written to Fc.

Comparisons are exact and never overflow or underflow. Four mutually exclusive relations are possible: less than, equal, greater than, and unordered. The unordered relation is true if one or both operands are NaN. (This behavior must be provided by a software trap handler, since NaNs trap.) Comparisons ignore the sign of zero, so +0 = –0.

Comparisons with plus and minus infinity execute normally and do not take an invalid operation trap.
Notes:

- In order to use CMPTxx with software completion trap handling, it is necessary to specify the /SU IEEE trap mode, even though an underflow trap is not possible.

- Compare Less Than A,B is the same as Compare Greater Than B,A; Compare Less Than or Equal A,B is the same as Compare Greater Than or Equal B,A. Therefore, only the less-than operations are included.
4.10.9 Convert VAX Floating to Integer

Format:

```
CVTGQ  Fb.rx,Fc.wq  !Floating-point Operate format
```

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

- Invalid Operation
- Integer Overflow

Instruction mnemonics:

```
CVTGQ    Convert G_floating to Quadword
```

Qualifiers:

- Rounding: Chopped (/C)
- Trapping: Software (/S)
  Integer Overflow Enable (/V)

Description:

The floating operand in register Fb is converted to a two's-complement quadword number and written to register Fc. The conversion aligns the operand fraction with the binary point just to the right of bit zero, rounds as specified, and complements the result if negative. Register Fa must be F31.

An invalid operation trap is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on integer overflow.
4.10.10 Convert Integer to VAX Floating

Format:

\[ \text{CVTQy Fb.rq,Fc.wx} \]

Floating-point Operate format

Operation:

\[ \text{Fc} \leftarrow \{\text{conversion of Fb}^{63:0}\} \]

Exceptions:

None

Instruction mnemonics:

\[ \text{CVTQF} \quad \text{Convert Quadword to F\_floating} \]
\[ \text{CVTQG} \quad \text{Convert Quadword to G\_floating} \]

Qualifiers:

Rounding: Chopped (/C)

Description:

The two's-complement quadword operand in register Fb is converted to a single- or double-precision floating result and written to register Fc. The conversion complements a number if negative, normalizes it, rounds to the target precision, and packs the result with an appropriate sign and exponent field. Register Fa must be F31.
4.10.11 Convert VAX Floating to VAX Floating

Format:

\[ \text{CVT}xy \quad \text{Fb}.rx,\text{Fc}.wx \]

!Floating-point Operate format

Operation:

\[ \text{Fc} \leftarrow \{\text{conversion of Fb}v\} \]

Exceptions:

Invalid Operation
Overflow
Underflow

Instruction mnemonics:

- CVTDG: Convert D\_floating to G\_floating
- CVTGD: Convert G\_floating to D\_floating
- CVTGF: Convert G\_floating to F\_floating

Qualifiers:

- Rounding: Chopped (/C)
- Trapping: Software (/S)
  Underflow Enable (/U)

Description:

The floating operand in register Fb is converted to the specified alternate floating format and written to register Fc. Register Fa must be F31.

An invalid operation trap is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on overflow or underflow.
Notes:

- The only arithmetic operations on D_floating values are conversions to and from G_floating. The conversion to G_floating rounds or chops as specified, removing three fraction bits. The conversion from G_floating to D_floating adds three low-order zeros as fraction bits, then the 8-bit exponent range is checked for overflow/underflow.

- The conversion from G_floating to F_floating rounds or chops to single precision, then the 8-bit exponent range is checked for overflow/underflow.

- No conversion from F_floating to G_floating is required, since F_floating values are always stored in registers as equivalent G_floating values.
4.10.12 Convert IEEE Floating to Integer

**Format:**

\[
\text{CVTTQ} \quad \text{Fb.rx,Fc.wq} \quad !\text{Floating-point Operate format}
\]

**Operation:**

\[
\text{Fc} \leftarrow \text{\{conversion of Fb\}}
\]

**Exceptions:**

Invalid Operation
Inexact Result
Integer Overflow

**Instruction mnemonics:**

CVTTQ Convert T_floating to Quadword

**Qualifiers:**

Rounding: Dynamic (/D)
          Minus infinity (/M)
          Chopped (/C)

Trapping:  Software (/S)
           Integer Overflow Enable (/V)
           Inexact Enable (/I)

**Description:**

The floating operand in register Fb is converted to a two's-complement number and written to register Fc. The conversion aligns the operand fraction with the binary point just to the right of bit zero, rounds as specified, and complements the result if negative. Register Fa must be F31.

See Section 4.7.6 for details of the stored result on integer overflow and inexact result.
4.10.13 Convert Integer to IEEE Floating

Format:

\[ \text{CVTQy} \ Fb.rq,Fc.wx \to \text{Floating-point Operate format} \]

Operation:

\[ Fc \leftarrow \{\text{conversion of } Fb<63:0>\} \]

Exceptions:

Inexact Result

Instruction mnemonics:

- CVTQS Convert Quadword to S\_floating
- CVTQT Convert Quadword to T\_floating

Qualifiers:

- Rounding: Dynamic (/D)
  - Minus infinity (/M)
  - Chopped (/C)
- Trapping: Software (/S)
  - Inexact Enable (/I)

Description:

The two's-complement operand in register Fb is converted to a single- or double-precision floating result and written to register Fc. The conversion complements a number if negative, normalizes it, rounds to the target precision, and packs the result with an appropriate sign and exponent field. Register Fa must be F31.

See Section 4.7.6 for details of the stored result on inexact result.

Notes:

- In order to use CVTQS or CVTQT with software completion trap handling, it is necessary to specify the /SUI IEEE trap mode, even though an underflow trap is not possible.
4.10.14 Convert IEEE S_Floating to IEEE T_Floating

Format:

```
CVTST Fb.rx,Fc.wx ! Floating-point Operate format
```

Operation:

```
Fc ← {conversion of Fbv}
```

Exceptions:

Invalid Operation

Instruction mnemonics:

```
CVTST Convert S_floating to T_floating
```

Qualifiers:

Trapping: Software (/S)

Description:

The S_floating operand in register Fb is converted to T_floating format and written to register Fc. Register Fa must be F31.

Notes:

- The conversion from S_floating to T_floating is exact. No rounding occurs. No underflow, overflow, or inexact result can occur. In fact, the conversion for finite values is the identity transformation.
- A trap handler can convert an S_floating denormal value into the corresponding T_floating finite value by adding 896 to the exponent and normalizing.
4.10.15 Convert IEEE T_Floating to IEEE S_Floating

Format:

```
CVTTS Fb.rx,Fc.wx !Floating-point Operate format
```

Operation:

```
Fc ← {conversion of Fb}v
```

Exceptions:

- Invalid Operation
- Overflow
- Underflow
- Inexact Result

Instruction mnemonics:

```
CVTTS Convert T_floating to S_floating
```

Qualifiers:

- Rounding: Dynamic (/D)
  - Minus infinity (/M)
  - Chopped (/C)
- Trapping: Software (/S)
  - Underflow Enable (/U)
  - Inexact Enable (/I)

Description:

The T_floating operand in register Fb is converted to S_floating format and written to register Fc. Register Fa must be F31.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.
4.10.16 VAX Floating Divide

Format:

\[ \text{DIVx} \quad \text{Fa.rx,Fb.rx,Fc.wx} \quad !\text{Floating-point Operate format} \]

Operation:

\[ Fc \leftarrow F_a \div F_b \]

Exceptions:

- Invalid Operation
- Division by Zero
- Overflow
- Underflow

Instruction mnemonics:

- DIVF Divide F_floating
- DIVG Divide G_floating

Qualifiers:

- Rounding: Chopped (/C)
- Trapping: Software (/S)
- Underflow Enable (/U)

Description:

The dividend operand in register \( F_a \) is divided by the divisor operand in register \( F_b \), and the quotient is written to register \( F_c \).

The quotient is rounded or chopped to the specified precision and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.
An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

A division by zero trap is signaled if Fbv is zero. The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on overflow or underflow.
4.10.17 IEEE Floating Divide

**Format:**

```
DIVx    Fa.rx,Fb.rx,Fc.wx    !Floating-point Operate format
```

**Operation:**

```
Fc ← Fav / Fbv
```

**Exceptions:**

- Invalid Operation
- Division by Zero
- Overflow
- Underflow
- Inexact Result

**Instruction mnemonics:**

- DIVS Divide S_floating
- DIVT Divide T_floating

**Qualifiers:**

- **Rounding:** Dynamic (/D)
  - Minus infinity (/M)
  - Chopped (/C)
- **Trapping:** Software (/S)
  - Underflow Enable (/U)
  - Inexact Enable (/I)

**Description:**

The dividend operand in register Fa is divided by the divisor operand in register Fb, and the quotient is written to register Fc.

The quotient is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.
4.10.18 Floating-Point Register to Integer Register Move

Format:

FTOIx Fa,rq,Rc,wq !Floating-point Operate format

Operation:

CASE:

FTOIS:
  Rc<63:32> ← SEXT(Fav<63>)
  Rc<31:0> ← Fav<63:62> || Fav <58:29>

FTOIT:
  Rc ← Fav
ENDCASE

Exceptions:

None

Instruction mnemonics:

  FTOIS  Floating-point to Integer Register Move, S_floating
  FTOIT  Floating-point to Integer Register Move, T_floating

Qualifiers:

None

Description:

Data in a floating-point register file is moved to an integer register file. The Fb field must be 31.

The instructions do not interpret bits in the register files; specifically, the instructions do not trap on non-finite values. Also, the instructions do not access memory.

FTOIS is exactly equivalent to the sequence:

  STS
  LDL

FTOIT is exactly equivalent to the sequence:

  STT
  LDQ
Software Note:

FTOIS and FTOIT are no slower than the corresponding store/load sequence, and can be significantly faster.
4.10.19 Integer Register to Floating-Point Register Move

Format:

ITOFx Ra.rq,Fc.wq !Floating-point Operate format

Operation:

CASE:

ITOFF:
Fc ← Rav<31> || MAP_F(Rav<30:23> || Rav<22:0> || 0<28:0>

ITOFS:
Fc ← Rav<31> || MAP_S(Rav<30:23> || Rav<22:0> || 0<28:0>

ITOFT:
Fc ← Rav

ENDCASE

Exceptions:

None

Instruction mnemonics:

ITOFF Integer to Floating-point Register Move, F_floating
ITOFS Integer to Floating-point Register Move, S_floating
ITOFT Integer to Floating-point Register Move, T_floating

Qualifiers:

None

Description:

Data in an integer register file is moved to a floating-point register file. The Rb field must be 31.

The instructions do not interpret bits in the register files; specifically, the instructions do not trap on non-finite values. Also, the instructions do not access memory.

ITOFF is equivalent to the following sequence, except that the word swapping that LDF normally performs is not performed by ITOFF:

STL
LDF
ITOFS is exactly equivalent to the sequence:

```
STL
LDS
```

ITOFT is exactly equivalent to the sequence:

```
STQ
LDT
```

**Software Note:**

ITOFF, ITOFS, and ITOFT are no slower than the corresponding store/load sequence, and can be significantly faster.
4.10.20 VAX Floating Multiply

Format:

\[
\text{MULx Fa.rx,Fb.rx,Fc.wx} \quad \text{!Floating-point Operate format}
\]

Operation:

\[
F_c \leftarrow F_a * F_b
\]

Exceptions:

- Invalid Operation
- Overflow
- Underflow

Instruction mnemonics:

- MULF Multiply F_floating
- MULG Multiply G_floating

Qualifiers:

- Rounding: Chopped (/C)
- Trapping: Software (/S)
  Underflow Enable (/U)

Description:

The multiplicand operand in register Fb is multiplied by the multiplier operand in register Fa, and the product is written to register Fc.

The product is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on overflow or underflow.
4.10.21 IEEE Floating Multiply

Format:

MULx Fa.rx,Fb.rx,Fc.wx !Floating-point Operate format

Operation:

Fc ← Fav * Fbv

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

MULS Multiply S_floating
MULT Multiply T_floating

Qualifiers:

Rounding: Dynamic (/D)
Minus infinity (/M)
Chopped (/C)

Trapping: Software (/S)
Underflow Enable (/U)
Inexact Enable (/I)

Description:

The multiplicand operand in register Fb is multiplied by the multiplier operand in register Fa, and the product is written to register Fc.

The product is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.
4.10.22 VAX Floating Square Root

Format:

\[
\text{SQRTx} \quad \text{Fb.wx,Fc.wx} \quad \text{!Floating-point Operate format}
\]

Operation:

\[
\text{Fc} \leftarrow \text{Fb}^{1/2}
\]

Exceptions:

Invalid operation

Instruction mnemonics:

\[
\begin{align*}
\text{SQRTF} & \quad \text{Square root F\text{\_floating}} \\
\text{SQRTG} & \quad \text{Square root G\text{\_floating}}
\end{align*}
\]

Qualifiers:

\[
\begin{align*}
\text{Rounding:} & \quad \text{Chopped (/C)} \\
\text{Trapping:} & \quad \text{Software (/S)} \\
\text{Underflow Enable (/U)} & \quad \text{!See Notes below}
\end{align*}
\]

Description:

The square root of the floating-point operand in register Fb is written to register Fc. (The Fa field of this instruction must be set to a value of 31.)

The result is rounded or chopped to the specified precision. The single-precision operation on a canonical single-precision value produces a canonical single-precision result.

An invalid operation is signaled if the operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). An invalid operation is signaled if the sign of the operand is negative.

The contents of the Fc are UNPREDICTABLE if an invalid operation is signaled.

Notes:

- Floating-point overflow and underflow are not possible for square root operation. The underflow enable qualifier is ignored.
4.10.23 IEEE Floating Square Root

Format:

\[
\text{SQRTx} \quad F_b.r_Fc.wx \quad \text{Floating-point Operate format}
\]

Operation:

\[
F_c \leftarrow F_b^{**} (1/2)
\]

Exceptions:

- Inexact result
- Invalid operation

Instruction mnemonics:

- SQRTS Square root S_floating
- SQRTT Square root T_floating

Qualifiers:

Rounding:
- Chopped (/C)
- Dynamic (/D)
- Minus infinity (/M)

Trapping:
- Inexact Result (/I)
- Software (/S)
- Underflow Enable (/U) !See Notes below

Description:

The square root of the floating-point operand in register Fb is written to register Fc. (The Fa field of this instruction must be set to a value of 31.)

The result is rounded to the specified precision. The single-precision operation on a canonical single-precision value produces a canonical single-precision result.

An invalid operation is signaled if the sign of the operand is less than zero. However, SQRT (–0) produces a result of –0.

Notes:

- Floating-point overflow and underflow are not possible for square root operation. The underflow enable qualifier is ignored.
4.10.24 VAX Floating Subtract

Format:

\[
\text{SUBx F.a.rx,F.b.rx,F.c.wx} \quad \text{!Floating-point Operate format}
\]

Operation:

\[
F_c \leftarrow F_a - F_b
\]

Exceptions:

- Invalid Operation
- Overflow
- Underflow

Instruction mnemonics:

- SUBF Subtract F\_floating
- SUBG Subtract G\_floating

Qualifiers:

- Rounding: Chopped (/C)
- Trapping: Software (/S)
- Underflow Enable (/U)

Description:

The subtrahend operand in register Fb is subtracted from the minuend operand in register Fa, and the difference is written to register Fc.

The difference is rounded or chopped to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.
An invalid operation trap is signaled if either operand has exp=0 and is not a true zero (that is, VAX reserved operands and dirty zeros trap). The contents of Fc are UNPREDICTABLE if this occurs.

See Section 4.7.6 for details of the stored result on overflow or underflow.
4.10.25 IEEE Floating Subtract

Format:

\[
\text{SUBx} \quad \text{Fa.rx,Fb.rx,Fc.wx} \quad \text{!Floating-point Operate format}
\]

Operation:

\[
\text{Fc} \leftarrow \text{Fav} - \text{Fbv}
\]

Exceptions:

Invalid Operation
Overflow
Underflow
Inexact Result

Instruction mnemonics:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBS</td>
<td>Subtract S_floating</td>
</tr>
<tr>
<td>SUBT</td>
<td>Subtract T_floating</td>
</tr>
</tbody>
</table>

Qualifiers:

Rounding: Dynamic (/D)
Minus infinity (/M)
Chopped (/C)

Trapping: Software (/S)
Underflow Enable (/U)
Inexact Enable (/I)

Description:

The subtrahend operand in register Fb is subtracted from the minuend operand in register Fa, and the difference is written to register Fc.

The difference is rounded to the specified precision, and then the corresponding range is checked for overflow/underflow. The single-precision operation on canonical single-precision values produces a canonical single-precision result.

See Section 4.7.6 for details of the stored result on overflow, underflow, or inexact result.
4.11 Miscellaneous Instructions

Alpha provides the miscellaneous instructions shown in Table 4–14.

Table 4–14: Miscellaneous Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMASK</td>
<td>Architecture Mask</td>
</tr>
<tr>
<td>CALL_PAL</td>
<td>Call Privileged Architecture Library Routine</td>
</tr>
<tr>
<td>ECB</td>
<td>Evict cache block</td>
</tr>
<tr>
<td>EXCB</td>
<td>Exception Barrier</td>
</tr>
<tr>
<td>FETCH</td>
<td>Prefetch Data</td>
</tr>
<tr>
<td>FETCH_M</td>
<td>Prefetch Data, Modify Intent</td>
</tr>
<tr>
<td>IMPLVER</td>
<td>Implementation Version</td>
</tr>
<tr>
<td>MB</td>
<td>Memory Barrier</td>
</tr>
<tr>
<td>RPCC</td>
<td>Read Processor Cycle Counter</td>
</tr>
<tr>
<td>TRAPB</td>
<td>Trap Barrier</td>
</tr>
<tr>
<td>WH64</td>
<td>Write hint—64 bytes</td>
</tr>
<tr>
<td>WMB</td>
<td>Write Memory Barrier</td>
</tr>
</tbody>
</table>
4.11.1 Architecture Mask

**Format:**

AMASK Rb.rq,Rc.wq !Operate format
AMASK #b.ib,Rc.wq !Operate format

**Operation:**

\[ Rc \leftarrow Rbv \text{ AND } \{\text{NOT CPU_feature_mask}\} \]

**Exceptions:**

None

**Instruction mnemonics:**

AMASK Architecture Mask

**Qualifiers:**

None

**Description:**

Rbv represents a mask of the requested architectural extensions. Bits are cleared that correspond to architectural extensions that are present. Reserved bits and bits that correspond to absent extensions are copied unchanged. In either case, the result is placed in Rc. If the result is zero, all requested features are present.

Software may specify an Rbv of all 1's to determine the complete set of architectural extensions implemented by a processor. Assigned bit definitions are located in Appendix D.

Ra must be R31 or the result in Rc is UNPREDICTABLE and it is UNPREDICTABLE whether an exception is signaled.

**Software Note:**

Use this instruction to make instruction-set decisions; use IMPLVER to make code-tuning decisions.

**Implementation Note:**

Instruction encoding is implemented as follows:

• On 21164 (EV5), AMASK copies Rbv to Rc.
• On 21164A (EV56), 21164PC (PCA56), and 21264 (EV6), AMASK correctly indicates support for architecture extensions by copying Rbv to Rc and clearing appropriate bits.

Bits are assigned and placed in Appendix D for architecture extensions, as ECOs for those extensions are passed. The low 8 bits are reserved for standard architecture extensions, so they can be tested with a literal; application-specific extensions are assigned from bit 8 upward.
4.11.2 Call Privileged Architecture Library

Format:

```
CALL_PAL  fnc.ir                      !PAL format
```

Operation:

{(Stall instruction issuing until all
 prior instructions are guaranteed to
 complete without incurring exceptions.)
 (Trap to PALcode.)}

Exceptions:

None

Instruction mnemonics:

```
CALL_PAL   Call Privileged Architecture Library
```

Qualifiers:

None

Description:

The CALL_PAL instruction is not issued until all previous instructions are
guaranteed to complete without exceptions. If an exception occurs, the continuation
PC in the exception stack frame points to the CALL_PAL instruction. The CALL_
PAL instruction causes a trap to PALcode.
4.11.3 Evict Data Cache Block

Format:

```
ECB   (Rb.ab) ! Misc format
```

Operation:

```
va ← Rbv
IF { va maps to memory space } THEN
  Prepare to reuse cache resources that are occupied by the
  the addressed byte.
END
```

Exceptions:

None

Instruction mnemonics:

```
ECB       Evict Cache Block
```

Qualifiers:

None

Description:

The ECB instruction provides a hint that the addressed location will not be referenced again in the near future, so any cache space it occupies should be made available to cache other memory locations. If the cache copy of the location is dirty, the processor may start writing it back; if the cache has multiple sets, the processor may arrange for the set containing the addressed byte to be the next set allocated.

The ECB instruction does not generate exceptions; if it encounters data address translation errors (access violation, translation not valid, and so forth) during execution, it is treated as a NOP.

If the address maps to non-memory-like (I/O) space, ECB is treated as a NOP.

Software Note:

ECB makes a particular cache location available for reuse by evicting and invalidating its contents. The intent is to give software more control over cache allocation policy in set-associative caches so that “useful” blocks can be retained in the cache.
ECB is a performance hint — it does not serialize the eviction of the addressed cache block with any preceding or following memory operation.

ECB is not intended for flushing caches prior to power failure or low power operation — CFLUSH is intended for that purpose.

Implementation Note:

Implementations with set-associative caches are encouraged to update their allocation pointer so that the next D-stream reference that misses the cache and maps to this line is allocated into the vacated set.
4.11.4 Exception Barrier

Format:

EXCB

! Memory format

Operation:

{EXCB does not appear to issue until completion of all exceptions and dependencies on the Floating-point Control Register (FPCR) from prior instructions.}

Exceptions:

None

Instruction mnemonics:

EXCB Exception Barrier

Qualifiers:

None

Description:

The EXCB instruction allows software to guarantee that in a pipelined implementation, all previous instructions have completed any behavior related to exceptions or rounding modes before any instructions after the EXCB are issued.

In particular, all changes to the Floating-point Control Register (FPCR) are guaranteed to have been made, whether or not there is an associated exception. Also, all potential floating-point exceptions and integer overflow exceptions are guaranteed to have been taken. EXCB is thus a superset of TRAPB.

If a floating-point exception occurs for which trapping is enabled, the EXCB instruction acts like a fault. In this case, the value of the Program Counter reported to the program may be the address of the EXCB instruction (or earlier), but is never the address of an instruction following the EXCB.

The relationship between EXCB and the FPCR is described in Section 4.7.7.1.
4.11.5 Prefetch Data

Format:

    FETCHx  0(Rb.ab)  !Memory format

Operation:

    va ← {Rbv}
    {Optionally prefetch aligned 512-byte block surrounding va.}

Exceptions:

    None

Instruction mnemonics:

    FETCH    Prefetch Data
    FETCH_M  Prefetch Data, Modify Intent

Qualifiers:

    None

Description:

The virtual address is given by Rbv. This address is used to designate an aligned 512-byte block of data. An implementation may optionally attempt to move all or part of this block (or a larger surrounding block) of data to a faster-access part of the memory hierarchy, in anticipation of subsequent Load or Store instructions that access that data.

The FETCH instruction is a hint to the implementation that may allow faster execution. An implementation is free to ignore the hint. If prefetching is done in an implementation, the order of fetch within the designated block is UNPREDICTABLE.

The FETCH_M instruction gives the additional hint that modifications (stores) to some or all of the data block are anticipated.

No exceptions are generated by FETCHx. If a Load (or Store in the case of FETCH_M) that uses the same address would fault, the prefetch request is ignored. It is UNPREDICTABLE whether a TB-miss fault is ever taken by FETCHx.
Implementation Note:

Implementations are encouraged to take the TB-miss fault, then continue the prefetch.

Software Note:

FETCH is intended to help software overlap memory latencies on the order of 100 cycles. FETCH is unlikely to help (or be implemented) for memory latencies on the order of 10 cycles. Code scheduling should be used to overlap such short latencies.

The programming model for effective use of FETCH and FETCH_M is given in Appendix A.
4.11.6 Implementation Version

Format:

```
IMPLVER Rc

!Operate format
```

Operation:

- Rc ← 0 for EV4, EV45, LCA, and LCA45 chips
  (21064, 21064A, 21066, 21068, 21066A)
- Rc ← 1 for EV5, EV56, and PCA56 chips
  (21164, 21164A, 21164PC)
- Rc ← 2 for EV6 and derivative chips
  (21264, etc.)

Exceptions:

None

Instruction mnemonics:

```
IMPLVER Implementation Version
```

Description:

A small integer is placed in Rc that specifies the major implementation version of the processor on which it is executed. This information can be used to make code-scheduling or tuning decisions, or the information can be used to branch to different pieces of code optimized for different implementations.

Note:

The value returned by IMPLVER does not identify the particular processor TYPE. Rather, it identifies a group of processors that can be treated similarly for performance characteristics such as scheduling. Ra must be R31 and Rb must be the literal #1 or the result in Rc is UNPREDICTABLE and it is UNPREDICTABLE whether an exception is signaled.

Software Note:

Use this instruction to make code-tuning decisions; use AMASK to make instruction-set decisions.

Values are assigned and placed in Appendix D.
4.11.7 Memory Barrier

Format:

\texttt{MB} !Memory format

Operation:

\{Guarantee that all subsequent loads or stores will not access memory until after all previous loads and stores have accessed memory, as observed by other processors.\}

Exceptions:

None

Instruction mnemonics:

\texttt{MB} Memory Barrier

Qualifiers:

None

Description:

The use of the Memory Barrier (MB) instruction is required only in multiprocessor systems. In the absence of an MB instruction, loads and stores to different physical locations are allowed to complete out of order on the issuing processor as observed by other processors. The MB instruction allows memory accesses to be serialized on the issuing processor as observed by other processors. See Chapter 5 for details on using the MB instruction to serialize these accesses. Chapter 5 also details coordinating memory accesses across processors.

Note that MB ensures serialization only; it does not necessarily accelerate the progress of memory operations.
4.11.8 Read Processor Cycle Counter

Format:

```
RPCC    Ra.wq !Memory format
```

Operation:

```
Ra ← {cycle counter}
```

Exceptions:

None

Instruction mnemonics:

```
RPCC    Read Processor Cycle Counter
```

Qualifiers:

None

Description:

Register Ra is written with the processor cycle counter (PCC). The PCC register consists of two 32-bit fields. The low-order 32 bits (PCC<31:0>) are an unsigned, wrapping counter, PCC_CNT. The high-order 32 bits (PCC<63:32>), PCC_OFF, are operating-system dependent in their implementation.

See Section 3.1.5 for a description of the PCC.

If an operating system uses PCC_OFF to calculate the per-process or per-thread cycle count, that count must be derived from the 32-bit sum of PCC_OFF and PCC_CNT. The following example computes that cycle count, modulo 2**32, and returns the count value in R0. Notice the care taken not to cause an unwanted sign extension.

```
RPCC R0 ; Read the process cycle counter
SLL R0, #32, R1 ; Line up the offset and count fields
ADDQ R0, R1, R0 ; Do add
SRL R0, #32, R0 ; Zero extend the cycle count to 64 bits
```

The following example code returns the value of PCC_CNT in R0<31:0> and all zeros in R0<63:32>.

```
RPCC R0
ZAPNOT R0,#15,R0
```
4.11.9 Trap Barrier

Format:

TRAPB !Memory format

Operation:

{TRAPB does not appear to issue until all prior instructions are guaranteed to complete without causing any arithmetic traps}.

Exceptions:

None

Instruction mnemonics:

TRAPB Trap Barrier

Qualifiers:

None

Description:

The TRAPB instruction allows software to guarantee that in a pipelined implementation, all previous arithmetic instructions will complete without incurring any arithmetic traps before the TRAPB or any instructions after it are issued.

If an arithmetic exception occurs for which trapping is enabled, the TRAPB instruction acts like a fault. In this case, the value of the Program Counter reported to the program may be the address of the TRAPB instruction (or earlier) but is never the address of the instruction following the TRAPB.

This fault behavior by TRAPB allows software, using one TRAPB instruction for each exception domain, to isolate the address range in which an exception occurs. If the address of the instruction following the TRAPB were allowed, there would be no way to distinguish an exception in the address range preceding a label from an exception in the range that includes the label along with the faulting instruction and a branch back to the label. This case arises when the code is not following software completion rules, but is inserting TRAPB instructions to isolate exceptions to the proper scope.

Use of TRAPB should be compared with use of the EXCB instruction; see Section 4.11.4.
4.11.10 Write Hint

Format:

```
WH64   (Rb.ab) ! Misc format
```

Operation:

```
va ← Rbv
IF { va maps to memory space } THEN
   Write UNPREDICTABLE data to the aligned 64-byte region
   containing the addressed byte.
END
```

Exceptions:

None

Instruction mnemonics:

```
WH64   Write Hint - 64 Bytes
```

Qualifiers:

None

Description:

The WH64 instruction provides a hint that the current contents of the aligned 64-byte block containing the addressed byte will never be read again, but will be overwritten in the near future.

The processor may allocate cache resources to hold the block without reading its previous contents from memory; the contents of the block may be set to any value that does not introduce a security hole, as described in Section 1.6.3.

The WH64 instruction does not generate exceptions; if it encounters data address translation errors (access violation, translation not valid, and so forth), it is treated as a NOP.

If the address maps to non-memory-like (I/O) space, WH64 is treated as a NOP.

Software Note:

This instruction is a performance hint that should be used when writing a large continuous region of memory. The intended code sequence consists of one WH64
instruction followed by eight quadword stores for each aligned 64-byte region to be written.

Sometimes, the UNPREDICTABLE data will exactly match some or all of the previous contents of the addressed block of memory.

Implementation Note:

If the 64-byte region containing the addressed byte is not in the data cache, implementations are encouraged to allocate the region in the data cache without first reading it from memory. However, if any of the addressed bytes exist in the caches of other processors, they must be kept coherent with respect to those processors.

Processors with cache blocks smaller than 64 bytes are encouraged to implement WH64 as defined. However, they may instead implement the instruction by allocating a smaller aligned cache block for write access or by treating WH64 as a NOP.

Processors with cache blocks larger than 64 bytes are also encouraged to implement WH64 as defined. However, they may instead treat WH64 as a NOP.
4.11.11 Write Memory Barrier

Format:

\[
\text{WMB} \quad \text{!Memory format}
\]

Operation:

\[
\{ \text{Guarantee that} \\
\{ \text{All preceding stores that access memory-like} \\
\{ \text{regions are ordered before any subsequent stores} \\
\{ \text{that access memory-like regions and} \\
\{ \text{All preceding stores that access non-memory-like} \\
\{ \text{regions are ordered before any subsequent stores} \\
\{ \text{that access non-memory-like regions.}
\]

Exceptions:

None

Instruction mnemonics:

- **WMB** Write Memory Barrier

Qualifiers:

None

Description:

The WMB instruction provides a way for software to control write buffers. It guarantees that writes preceding the WMB are not aggregated with writes that follow the WMB.

WMB guarantees that writes to memory-like regions that precede the WMB are ordered before writes to memory-like regions that follow the WMB. Similarly, WMB guarantees that writes to non-memory-like regions that precede the WMB are ordered before writes to non-memory-like regions that follow the WMB. It does not order writes to memory-like regions relative to writes to non-memory-like regions.

WMB causes writes that are contained in buffers to be completed without unnecessary delay. It is particularly suited for batching writes to high-performance I/O devices.

WMB prevents writes that precede the WMB from being merged with writes that follow the WMB. In particular, two writes that access the same location and are separated by a WMB cause two distinct and ordered write events.
In the absence of a WMB (or IMB or MB) instruction, stores to memory-like or non-memory-like regions can be aggregated and/or buffered and completed in any order.

The WMB instruction is the preferred method for providing high-bandwidth write streams where order must be preserved between writes in that stream.

Notes:
WMB is useful for ordering streams of writes to a non-memory-like region, such as to memory-mapped control registers or to a graphics frame buffer. While both MB and WMB can ensure that writes to a non-memory-like region occur in order, without being aggregated or reordered, the WMB is usually faster and is never slower than MB.

WMB can correctly order streams of writes in programs that operate on shared sections of data if the data in those sections are protected by a classic semaphore protocol. The following example illustrates such a protocol:

<table>
<thead>
<tr>
<th>Processor i</th>
<th>Processor j</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Acquire lock&gt;</td>
<td>&lt;Acquire lock&gt;</td>
</tr>
<tr>
<td>MB</td>
<td>MB</td>
</tr>
<tr>
<td>&lt;Read and write data in shared section&gt;</td>
<td>&lt;Read and write data in shared section&gt;</td>
</tr>
<tr>
<td>WMB</td>
<td>WMB</td>
</tr>
<tr>
<td>&lt;Release lock&gt;</td>
<td>⇒</td>
</tr>
</tbody>
</table>

The example above is similar to that in Section 5.5.3, except a WMB is substituted for the second MB in the lock-update-release sequence. It is correct to substitute WMB for the second MB only if:

A. All data locations that are read or written in the critical section are accessed only after acquiring a software lock by using lock_variable (and before releasing the software lock).

B. For each read $u$ of shared data in the critical section, there is a write $v$ such that:
   1. $v$ is BEFORE the WMB
   2. $v$ follows $u$ in processor issue sequence (see Section 5.6.1.1)
   3. $v$ either depends on $u$ (see Section 5.6.1.7) or overlaps $u$ (see Section 5.6.1), or both.

C. Both lock_variable and all the shared data are in memory-like regions (or lock_variable and all the shared data are in non-memory-like regions). If the lock_
variable is in a non-memory-like region, the atomic lock protocol must use some implementation-specific hardware support.

The substitution of a WMB for the second MB is usually faster and never slower.
4.12 VAX Compatibility Instructions

Alpha provides the instructions shown in Table 4-15 for use in translated VAX code. These instructions are not a permanent part of the architecture and will not be available in some future implementations. They are intended to preserve customer assumptions about VAX instruction atomicity in porting code from VAX to Alpha.

These instructions should be generated only by the VAX-to-Alpha software translator; they should never be used in native Alpha code. Any native code that uses them may cease to work.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>Read and Clear</td>
</tr>
<tr>
<td>RS</td>
<td>Read and Set</td>
</tr>
</tbody>
</table>
4.12.1 VAX Compatibility Instructions

Format:

\[
\begin{align*}
R_x & \quad \text{Ra.wq} & \quad !\text{Memory format}
\end{align*}
\]

Operation:

\[
\begin{align*}
\text{Ra} & \leftarrow \text{intr\_flag} \\
\text{intr\_flag} & \leftarrow 0 & !\text{RC} \\
\text{intr\_flag} & \leftarrow 1 & !\text{RS}
\end{align*}
\]

Exceptions:

None

Instruction mnemonics:

- RC: Read and Clear
- RS: Read and Set

Qualifiers:

None

Description:

The intr\_flag is returned in Ra and then cleared to zero (RC) or set to one (RS).

These instructions may be used to determine whether the sequence of Alpha instructions between RS and RC (corresponding to a single VAX instruction) was executed without interruption or exception.

Intr\_flag is a per-processor state bit. The intr\_flag is cleared if that processor encounters a CALL_PAL REI instruction.

It is UNPREDICTABLE whether a processor’s intr\_flag is affected when that processor executes an LDx_L or STx_C instruction. A processor’s intr\_flag is not affected when that processor executes a normal load or store instruction.

A processor’s intr\_flag is not affected when that processor executes a taken branch.

Note:

These instructions are intended only for use by the VAX-to-Alpha software translator; they should never be used by native code.
4.13 Multimedia (Graphics and Video) Support Instructions

Alpha provides the following instructions that enhance support for graphics and video algorithms:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINUB8</td>
<td>Vector Unsigned Byte Minimum</td>
</tr>
<tr>
<td>MINSB8</td>
<td>Vector Signed Byte Minimum</td>
</tr>
<tr>
<td>MINUW4</td>
<td>Vector Unsigned Word Minimum</td>
</tr>
<tr>
<td>MINSW4</td>
<td>Vector Signed Word Minimum</td>
</tr>
<tr>
<td>MAXUB8</td>
<td>Vector Unsigned Byte Maximum</td>
</tr>
<tr>
<td>MAXSB8</td>
<td>Vector Signed Byte Maximum</td>
</tr>
<tr>
<td>MAXUW4</td>
<td>Vector Unsigned Word Maximum</td>
</tr>
<tr>
<td>MAXSW4</td>
<td>Vector Signed Word Maximum</td>
</tr>
<tr>
<td>PERR</td>
<td>Pixel Error</td>
</tr>
<tr>
<td>PKLB</td>
<td>Pack Longwords to Bytes</td>
</tr>
<tr>
<td>PKWB</td>
<td>Pack Words to Bytes</td>
</tr>
<tr>
<td>UNPKBL</td>
<td>Unpack Bytes to Longwords</td>
</tr>
<tr>
<td>UNPKBW</td>
<td>Unpack Bytes to Words</td>
</tr>
</tbody>
</table>

The MIN and MAX instructions allow the clamping of pixel values to maximum values that are allowed in different standards and stages of the CODECs.

The PERR instruction accelerates the macroblock search in motion estimation.

The pack and unpack (PKxB and UNPKBx) instructions accelerate the blocking of interleaved YUV coordinates for processing by the CODEC.
4.13.1 Byte and Word Minimum and Maximum

Format:

\[
\begin{align*}
\text{MINxxx} & \quad \text{Ra.rq,Rb.rq,Rc.wq} & \text{! Operate Format} \\
\text{MAXxxx} & \quad \text{Ra.rq,#b.ib,Rc.wq} & \text{! Operate Format}
\end{align*}
\]

Operation:

CASE

\begin{align*}
\text{MINUB8:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 7 \\
& \quad \text{Rcv}<i*8+7:i*8> = \text{MINU}(\text{Rav}<i*8+7:i*8>,\text{Rbv}<i*8+7:i*8>) \\
& \quad \text{END} \\
\text{MINSB8:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 7 \\
& \quad \text{Rcv}<i*8+7:i*8> = \text{MINS}(\text{Rav}<i*8+7:i*8>,\text{Rbv}<i*8+7:i*8>) \\
& \quad \text{END} \\
\text{MINUW4:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 3 \\
& \quad \text{Rcv}<i*16+15:i*16> = \text{MINU}(\text{Rav}<4i*16+15:i*16>,\text{Rbv}<i*16+15:i*16>) \\
& \quad \text{END} \\
\text{MINSW4:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 3 \\
& \quad \text{Rcv}<i*16+15:i*16> = \text{MINS}(\text{Rav}<i*16+15:i*16>,\text{Rbv}<i*16+15:i*16>) \\
& \quad \text{END} \\
\text{MAXUB8:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 7 \\
& \quad \text{Rcv}<i*8+7:i*8> = \text{MAXU}(\text{Rav}<i*8+7:i*8>,\text{Rbv}<i*8+7:i*8>) \\
& \quad \text{END} \\
\text{MAXSB8:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 7 \\
& \quad \text{Rcv}<i*8+7:i*8> = \text{MAXS}(\text{Rav}<i*8+7:i*8>,\text{Rbv}<i*8+7:i*8>) \\
& \quad \text{END} \\
\text{MAXUW4:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 3 \\
& \quad \text{Rcv}<i*16+15:i*16> = \text{MAXU}(\text{Rav}<i*16+15:i*16>,\text{Rbv}<i*16+15:i*16>) \\
& \quad \text{END} \\
\text{MAXSW4:} & \quad \text{FOR } i \text{ FROM } 0 \text{ TO } 3 \\
& \quad \text{Rcv}<i*16+15:i*16> = \text{MAXS}(\text{Rav}<i*16+15:i*16>,\text{Rbv}<i*16+15:i*16>) \\
& \quad \text{END}
\end{align*}

ENDCASE:

Exceptions:

None
Instruction mnemonics:

- MINUB8   Vector Unsigned Byte Minimum
- MINSB8   Vector Signed Byte Minimum
- MINUW4   Vector Unsigned Word Minimum
- MINSW4   Vector Signed Word Minimum
- MAXUB8   Vector Unsigned Byte Maximum
- MAXSB8   Vector Signed Byte Maximum
- MAXUW4   Vector Unsigned Word Maximum
- MAXSW4   Vector Signed Word Maximum

Qualifiers:

- None

Description:

For MINxB8, each byte of Rc is written with the smaller of the corresponding bytes of Ra or Rb. The bytes may be interpreted as signed or unsigned values.

For MINxW4, each word of Rc is written with the smaller of the corresponding words of Ra or Rb. The words may be interpreted as signed or unsigned values.

For MAXxB8, each byte of Rc is written with the larger of the corresponding bytes of Ra or Rb. The bytes may be interpreted as signed or unsigned values.

For MAXxW4, each word of Rc is written with the larger of the corresponding words of Ra or Rb. The words may be interpreted as signed or unsigned values.
4.13.2 Pixel Error

Format:

```
PERR Ra.rq,Rb.rq,Rc.wq ! Operate Format
```

Operation:

```
temp = 0
FOR i FROM 0 TO 7
  IF { Rav<i*8+7:i*8> GEU Rbv<i*8+7:i*8>} THEN
    temp ← temp + (Rav<i*8+7:i*8> - Rbv<i*8+7:i*8>)
  ELSE
    temp ← temp + (Rbv<i*8+7:i*8> - Rav<i*8+7:i*8>)
  END
END
Rc ← temp
```

Exceptions:

None

Instruction mnemonics:

```
PERR Pixel Error
```

Qualifiers:

None

Description:

The absolute value of the difference between each of the bytes in Ra and Rb is calculated. The sum of the resulting bytes is written to Rc.
4.13.3 Pack Bytes

Format:

\[ PKxB \quad Rb.rq,Rc.wq \quad \text{! Operate Format} \]

Operation:

\[
\text{CASE}
\] PKLB:
\[
\text{BEGIN}
Rc<07:00> \leftarrow Rbv<07:00>
Rc<15:08> \leftarrow Rbv<39:32>
Rc<63:16> \leftarrow 0
\text{END}
\]

PKWB:
\[
\text{BEGIN}
Rc<07:00> \leftarrow Rbv<07:00>
Rc<15:08> \leftarrow Rbv<23:16>
Rc<23:16> \leftarrow Rbv<39:32>
Rc<31:24> \leftarrow Rbv<55:48>
Rc<63:32> \leftarrow 0
\text{END}
\]

ENDCASE

Exceptions:

None

Instruction mnemonics:

PKLB Pack Longwords to Bytes
PKWB Pack Words to Bytes

Qualifiers:

None

Description:

For PKLB, the component longwords of Rb are truncated to bytes and written to the lower two byte positions of Rc. The upper six bytes of Rc are written with zero.

For PKWB, the component words of Rb are truncated to bytes and written to the lower four byte positions of Rc. The upper four bytes of Rc are written with zero.
4.13.4 Unpack Bytes

Format:

UNPKBx  Rb_rq,Rc_wq  ! Operate Format

Operation:

\[
\begin{align*}
t & = 0 \\
\text{CASE} \\
\text{UNPKBL:} & \\
\text{BEGIN} & \\
\text{temp}<07:00> & = \text{Rbv}<07:00> \\
\text{temp}<39:32> & = \text{Rbv}<15:08> \\
\text{END} & \\
\text{UNPKBW:} & \\
\text{BEGIN} & \\
\text{temp}<07:00> & = \text{Rbv}<07:00> \\
\text{temp}<23:16> & = \text{Rbv}<15:08> \\
\text{temp}<39:32> & = \text{Rbv}<23:16> \\
\text{temp}<55:48> & = \text{Rbv}<31:24> \\
\text{END} & \\
\text{ENDCASE} \\
\text{Rc} & \leftarrow \text{temp}
\end{align*}
\]

Exceptions:

None

Instruction mnemonics:

UNPKBL  Unpack Bytes to Longwords
UNPKBW  Unpack Bytes to Words

Qualifiers:

None

Description:

For UNPKBL, the lower two component bytes of Rb are zero-extended to longwords. The resulting longwords are written to Rc.

For UNPKBW, the lower four component bytes of Rb are zero-extended to words. The resulting words are written to Rc.
5.1 Introduction

Portions of the Alpha architecture have implications for programming, and the system structure, of both uniprocessor and multiprocessor implementations. Architectural implications considered in the following sections are:

- Physical address space behavior
- Caches and write buffers
- Translation buffers and virtual caches
- Data sharing
- Read/write ordering
- Arithmetic traps

To meet the requirements of the Alpha architecture, software and hardware implementors need to take these issues into consideration.

5.2 Physical Address Space Characteristics

Alpha physical address space is divided into four equal-size regions. The regions are delineated by the two most significant, implemented, physical address bits. Each region's characteristics are distinguished by the coherency, granularity, and width of memory accesses, and whether the region exhibits memory-like behavior or non-memory-like behavior.

5.2.1 Coherency of Memory Access

Alpha implementations must provide a coherent view of memory, in which each write by a processor or I/O device (hereafter, called “processor”) becomes visible to all other processors. No distinction is made between coherency of “memory space” and “I/O space”.

Memory coherency may be provided in different ways, for each of the four physical address regions.

Possible per-region policies include, but are not restricted to:

1. No caching
No copies are kept of data in a region; all reads and writes access the actual data location (memory or I/O register), but a processor may elide multiple accesses to the same data (see Section 5.2.3).

2. Write-through caching
Copies are kept of any data in the region; reads may use the copies, but writes update the actual data location and either update or invalidate all copies.

3. Write-back caching
Copies are kept of any data in the region; reads and writes may use the copies, and writes use additional state to determine whether there are other copies to invalidate or update.

Software/Hardware Note:
To produce separate and distinct accesses to a specific location, the location must be a region with no caching and a memory barrier instruction must be inserted between accesses. See Section 5.2.3.

Part of the coherency policy implemented for a given physical address region may include restrictions on excess data transfers (performing more accesses to a location than is necessary to acquire or change the location's value), or may specify data transfer widths (the granularity used to access a location).

Independent of coherency policy, a processor may use different hardware or different hardware resource policies for caching or buffering different physical address regions.

5.2.2 Granularity of Memory Access
For each region, an implementation must support aligned quadword access and may optionally support aligned longword access or byte access. If byte access is supported in a region, aligned word access and aligned longword access are also supported.

For a quadword access region, accesses to physical memory must be implemented such that independent accesses to adjacent aligned quadwords produce the same results regardless of the order of execution. Further, an access to an aligned quadword must be done in a single atomic operation.

For a longword access region, accesses to physical memory must be implemented such that independent accesses to adjacent aligned longwords produce the same results regardless of the order of execution. Further, an access to an aligned longword must be done in a single atomic operation, and an access to an aligned quadword must also be done in a single atomic operation.

For a byte access region, accesses to physical memory must be implemented such that independent accesses to adjacent bytes or adjacent aligned words produce the same results, regardless of the order of execution. Further, an access to a byte, an aligned word, an aligned longword, or an aligned quadword must be done in a single atomic operation.
In this context, “atomic” means that the following is true if different processors do simultaneous reads and writes of the same data:

- The result of any set of writes must be the same as if the writes had occurred sequentially in some order, and

- Any read that observes the effect of a write on some part of memory must observe the effect of that write (or of a later write or writes) on the entire part of memory that is accessed by both the read and the write.

When a write accesses only part of a given word, longword, or quadword, a read of the entire structure may observe the effect of that partial write without observing the effect of an earlier write of another byte or bytes to the same structure. See Sections 5.6.1.5 and 5.6.1.6.

### 5.2.3 Width of Memory Access

Subject to the granularity, ordering, and coherency constraints given in Sections 5.2.1, 5.2.2, and 5.6, accesses to physical memory may be freely cached, buffered, and prefetched.

A processor may read more physical memory data (such as a full cache block) than is actually accessed, writes may trigger reads, and writes may write back more data than is actually updated. A processor may elide multiple reads and/or writes to the same data.

### 5.2.4 Memory-Like and Non-Memory-Like Behavior

Memory-like regions obey the following rules:

- Each page frame in the region either exists in its entirety or does not exist in its entirety; there are no holes within a page frame.

- All locations that exist are read/write.

- A write to a location followed by a read from that location returns precisely the bits written; all bits act as memory.

- A write to one location does not change any other location.

- Reads have no side effects.

- Longword access granularity is provided, and if the byte/word extension is implemented, byte access granularity is provided.

- Instruction-fetch is supported.

- Load-locked and store-conditional are supported.

Non-memory-like regions may have much more arbitrary behavior:

- Unimplemented locations or bits may exist anywhere.

- Some locations or bits may be read-only and others write-only.

- Address ranges may overlap, such that a write to one location changes the bits read from a different location.
• Reads may have side effects, although this is strongly discouraged.
• Longword granularity need not be supported and, even if the byte/word extension is implemented, byte access granularity need not be implemented.
• Instruction-fetch need not be supported.
• Load-locked and store-conditional need not be supported.

Hardware/Software Coordination Note:
The details of such behavior are outside the scope of the Alpha architecture. Specific processor and I/O device implementations may choose and document whatever behavior they need. It is the responsibility of system designers to impose enough consistency to allow processors successfully to access matching non-memory devices in a coherent way.

5.3 Translation Buffers and Virtual Caches
A system may choose to include a virtual instruction cache (virtual I-cache) or a virtual data cache (virtual D-cache). A system may also choose to include either a combined data and instruction translation buffer (TB) or separate data and instruction TBs (DTB and ITB). The contents of these caches and/or translation buffers may become invalid, depending on what operating system activity is being performed.

Whenever a non-software field of a valid page table entry (PTE) is modified, copies of that PTE must be made coherent. PALcode mechanisms are available to clear all TBs, both DTB and ITB entries for a given VA, either DTB or ITB entries for a given VA, or all entries with the address space match (ASM) bit clear. Virtual D-cache entries are made coherent whenever the corresponding DTB entry is requested to be cleared by any of the appropriate PALcode mechanisms. Virtual I-cache entries can be made coherent via the CALL_PAL IMB instruction.

If a processor implements address space numbers (ASNs), and the old PTE has the Address Space Match (ASM) bit clear (ASNs in use) and the Valid bit set, then entries can also effectively be made coherent by assigning a new, unused ASN to the currently running process and not reusing the previous ASN before calling the appropriate PALcode routine to invalidate the translation buffer (TB).

In a multiprocessor environment, making the TBs and/or caches coherent on only one processor is not always sufficient. An operating system must arrange to perform the above actions on each processor that could possibly have copies of the PTE or data for any affected page.

5.4 Caches and Write Buffers
A hardware implementation may include mechanisms to reduce memory access time by making local copies of recently used memory contents (or those expected to be used) or by buffering writes to complete at a later time. Caches and write buffers are examples of these mechanisms. They must be implemented so that their existence
is transparent to software (except for timing, error reporting/control/recovery, and modification to the I-stream).

The following requirements must be met by all cache/write-buffer implementations. All processors must provide a coherent view of memory.

1. Write buffers may be used to delay and aggregate writes. From the viewpoint of another processor, buffered writes appear not to have happened yet. (Write buffers must not delay writes indefinitely. See Section 5.6.1.9.)

2. Write-back caches must be able to detect a later write from another processor and invalidate or update the cache contents.

3. A processor must guarantee that a data store to a location followed by a data load from the same location reads the updated value.

4. Cache prefetching is allowed, but virtual caches must not prefetch from invalid pages. See Sections 5.6.1.3, 5.6.4.3, and 5.6.4.4.

5. A processor must guarantee that all of its previous writes are visible to all other processors before a HALT instruction completes. A processor must guarantee that its caches are coherent with the rest of the system before continuing from a HALT.

6. If battery backup is supplied, a processor must guarantee that the memory system remains coherent across a powerfail/recovery sequence. Data that was written by the processor before the powerfail may not be lost, and any caches must be in a valid state before (and if) normal instruction processing is continued after power is restored.

7. Virtual instruction caches are not required to notice modifications of the virtual I-stream (they need not be coherent with the rest of memory). Software that creates or modifies the instruction stream must execute a CALL_PAL IMB before trying to execute the new instructions.

In this context, to “modify the virtual I-stream” means any Store to the same physical address that is subsequently fetched as an instruction via some corresponding (virtual address, ASN) pair, or to change the virtual-to-physical address mapping so that different values are fetched.

For example, if two different virtual addresses, VA1 and VA2, map to the same page frame, a store to VA1 modifies the virtual I-stream fetched via VA2.

However, the following sequence does not modify the virtual I-stream (this might happen in soft page faults).

1. Change the mapping of an I-stream page from valid to invalid.
2. Copy the corresponding page frame to a new page frame.
3. Change the original mapping to be valid and point to the new page frame.

8. Physical instruction caches are not required to notice modifications of the physical I-stream (they need not be coherent with the rest of memory), except for certain paging activity. (See Section 5.6.4.4.) Software that creates or modifies
the instruction stream must execute a CALL_PAL IMB before trying to execute the new instructions.

In this context, to “modify the physical I-stream” means any Store to the same physical address that is subsequently fetched as an instruction.

5.5 Data Sharing

In a multiprocessor environment, writes to shared data must be synchronized by the programmer.

5.5.1 Atomic Change of a Single Datum

The ordinary STL and STQ instructions can be used to perform an atomic change of a shared aligned longword or quadword. (“Change” means that the new value is not a function of the old value.) In particular, an ordinary STL or STQ instruction can be used to change a variable that could be simultaneously accessed via an LDx_L/STx_C sequence.

5.5.2 Atomic Update of a Single Datum

The load-locked/store-conditional instructions may be used to perform an atomic update of a shared aligned longword or quadword. (“Update” means that the new value is a function of the old value.)

The following sequence performs a read-modify-write operation on location \( x \). Only register-to-register operate instructions and branch fall-throughs may occur in the sequence:

\[
\text{try_again:}
\begin{align*}
\text{LDQ}_L & \quad R1,x \\
& \quad <\text{modify } R1> \\
\text{STQ}_C & \quad R1,x \\
\text{BEQ} & \quad R1,\text{no\_store} \\
& : \\
\text{no\_store:} & \\
& \quad <\text{code to check for excessive iterations}> \\
& \quad \text{BR} \quad \text{try\_again}
\end{align*}
\]

If this sequence runs with no exceptions or interrupts, and no other processor writes to location \( x \) (more precisely, the locked range including \( x \)) between the LDQ_L and STQ_C instructions, then the STQ_C shown in the example stores the modified value in \( x \) and sets R1 to 1. If, however, the sequence encounters exceptions or interrupts that eventually continue the sequence, or another processor writes to \( x \), then the STQ_C does not store and sets R1 to 0. In this case, the sequence is repeated via the branches to no_store and try_again. This repetition continues until the reasons for exceptions or interrupts are removed, and no interfering store is encountered.

To be useful, the sequence must be constructed so that it can be replayed an arbitrary number of times, giving the same result values each time. A sufficient (but not necessary) condition is that, within the sequence, the set of operand destinations and the set of operand sources are disjoint.
Note:
A sufficiently long instruction sequence between LDx_L and STx_C will never complete, because periodic timer interrupts will always occur before the sequence completes. The rules in Appendix A describe sequences that will eventually complete in all Alpha implementations.

This load-locked/store-conditional paradigm may be used whenever an atomic update of a shared aligned quadword is desired, including getting the effect of atomic byte writes.

5.5.3 Atomic Update of Data Structures

Before accessing shared writable data structures (those that are not a single aligned longword or quadword), the programmer can acquire control of the data structure by using an atomic update to set a software lock variable. Such a software lock can be cleared with an ordinary store instruction.

A software-critical section, therefore, may look like the sequence:

```
stq_c_loop:
spin_loop:
  LDQ   R1,lock_variable ; This optional spin-loop code
  BLBS  R1,already_set  ; should be used unless the
                        ; lock is known to be low-contention.
  LDQ_L R1,lock_variable ;  
  BLBS  R1,already_set  ;  
  OR    R1,#1,R2        ;  > Set lock bit
  STQ_C R2,lock_variable ; / 
  BEQ   R2,stq_c_fail  ; / 
  MB
  <critical section: updates various data structures>
MB ; Second MB
  STQ   R31,lock_variable ; Clear lock bit
  :
  already_set:
    <code to block or reschedule or test for too many iterations>
BR   spin_loop
stq_c_fail:
    <code to test for too many iterations>
BR   stq_c_loop
```

This code has a number of subtleties:

1. If the lock_variable is already set, the spin loop is done without doing any stores. This avoidance of stores improves memory subsystem performance and avoids the deadlock described below. The loop uses an ordinary load. This code sequence is preferred unless the lock is known to be low-contention, because the sequence increases the probability that the LDQ_L hits in the cache and the LDQ_L/STQ_C sequence completes quickly and successfully.
2. If the lock_variable is actually being changed from 0 to 1, and the STQ_C fails (due to an interrupt, or because another processor simultaneously changed lock_variable), the entire process starts over by reading the lock_variable again.

3. Only the fall-through path of the BLBS instructions do a STx_C; some implementations may not allow a successful STx_C after a branch-taken.

4. Only register-to-register operate instructions are used to do the modify.

5. Both conditional branches are forward branches, so they are properly predicted not to be taken (to match the common case of no contention for the lock).

6. The OR writes its result to a second register; this allows the OR and the BLBS to be interchanged if that would give a faster instruction schedule.

7. Other operate instructions (from the critical section) may be scheduled into the LDQ_L..STQ_C sequence, so long as they do not fault or trap, and they give correct results if repeated; other memory or operate instructions may be scheduled between the STQ_C and BEQ.

8. The memory barrier instructions are discussed in Section 5.5.4. It is correct to substitute WMB for the second MB only if:

   A. All data locations that are read or written in the critical section are accessed only after acquiring a software lock by using lock_variable (and before releasing the software lock).

   B. For each read u of shared data in the critical section, there is a write v such that:

      1. v is BEFORE the WMB
      2. v follows u in processor issue sequence (see Section 5.6.1.1)
      3. v either depends on u (see Section 5.6.1.7) or overlaps u (see Section 5.6.1), or both.

   C. Both lock_variable and all the shared data are in memory-like regions (or lock_variable and all the shared data are in non-memory-like regions). If the lock_variable is in a non-memory-like region, the atomic lock protocol must use some implementation-specific hardware support.

   Generally, the substitution of a WMB for the second MB increases performance.

9. An ordinary STQ instruction is used to clear the lock_variable.

It would be a performance mistake to spin-wait by repeating the full LDQ_L..STQ_C sequence (to move the BLBS after the BEQ) because that sequence may repeatedly change the software lock_variable from “locked” to “locked,” with each write causing extra access delays in all other caches that contain the lock_variable. In the extreme, spin-waits that contain writes may deadlock as follows:

   If, when one processor spins with writes, another processor is modifying (not changing) the lock_variable, then the writes on the first processor may cause the STx_C of the modify on the second processor always to fail.
This deadlock situation is avoided by:

- Having only one processor execute a store (no STx_C), or
- Having no write in the spin loop, or
- Doing a write only if the shared variable actually changes state (1 → 1 does not change state).

5.5.4 Ordering Considerations for Shared Data Structures

A critical section sequence, such as shown in Section 5.5.3, is conceptually only three steps:

1. Acquire software lock
2. Critical section—read/write shared data
3. Clear software lock

In the absence of explicit instructions to the contrary, the Alpha architecture allows reads and writes to be reordered. While this may allow more implementation speed and overlap, it can also create undesired side effects on shared data structures. Normally, the critical section just described would have two instructions added to it:

```
<acquire software lock>
MB (memory barrier #1)
<critical section -- read/write shared data>
MB (memory barrier #2)
<clear software lock>
```

The first memory barrier prevents any reads (from within the critical section) from being prefetched before the software lock is acquired; such prefetched reads would potentially contain stale data.

The second memory barrier prevents any writes and reads in the critical section being delayed past the clearing of the software lock. Such delayed accesses could interact with the next user of the shared data, defeating the purpose of the software lock entirely. It is correct to substitute WMB for the second MB only if:

- All data locations that are read or written in the critical section are accessed only after acquiring a software lock by using lock_variable (and before releasing the software lock).

- For each read u of shared data in the critical section, there is a write v such that:
  1. v is BEFORE the WMB
  2. v follows u in processor issue sequence (see Section 5.6.1.1)
  3. v either depends on u (see Section 5.6.1.7) or overlaps u (see Section 5.6.1), or both.

- Both lock_variable and all the shared data are in memory-like regions (or lock_variable and all the shared data are in non-memory-like regions). If the lock_
variable is in a non-memory-like region, the atomic lock protocol must use some implementation-specific hardware support.

Generally, the substitution of a WMB for the second MB increases performance.

Software Note:

In the VAX architecture, many instructions provide noninterruptable read-modify-write sequences to memory variables. Most programmers never regard data sharing as an issue.

In the Alpha architecture, programmers must pay more attention to synchronizing access to shared data; for example, to AST routines. In the VAX, a programmer can use an ADDL2 to update a variable that is shared between a “MAIN” routine and an AST routine, if running on a single processor. In the Alpha architecture, a programmer must deal with AST shared data by using multiprocessor shared data sequences.

5.6 Read/Write Ordering

This section applies to programs that run on multiple processors or on one or more processors that are interacting with DMA I/O devices. To a program running on a single processor and not interacting with DMA I/O devices, all memory accesses appear to happen in the order specified by the programmer. This section deals with predictable read/write ordering across multiple processors and/or DMA I/O devices.

The order of reads and writes done in an Alpha implementation may differ from that specified by the programmer.

For any two memory accesses A and B, either A must occur before B in all Alpha implementations, B must occur before A, or they are UNORDERED. In the last case, software cannot depend upon one occurring first: the order may vary from implementation to implementation, and even from run to run or moment to moment on a single implementation.

If two accesses cannot be shown to be ordered by the rules given, they are UNORDERED and implementations are free to do them in any order that is convenient. Implementations may take advantage of this freedom to deliver substantially higher performance.

The discussion that follows first defines the architectural issue sequence of memory accesses on a single processor, then defines the (partial) ordering on this issue sequence that all Alpha implementations are required to maintain.

The individual issue sequences on multiple processors are merged into access sequences at each shared memory location. The discussion defines the (partial) ordering on the individual access sequences that all Alpha implementations are required to maintain.

The net result is that for any code that executes on multiple processors, one can determine which memory accesses are required to occur before others on all Alpha implementations and hence can write useful shared-variable software.
Software writers can force one access to occur before another by inserting a memory barrier instruction (MB, WMB, or CALL_PAL IMB) between the accesses.

### 5.6.1 Alpha Shared Memory Model

An Alpha system consists of a collection of processors, I/O devices (and possibly a bridge to connect remote I/O devices), and shared memories that are accessible by all processors.

**Note:**

An example of an unshared location is a physical address in I/O space that refers to a CSR that is local to a processor and not accessible by other processors.

A processor is an Alpha CPU.

In most systems, DMA I/O devices or other agents can read or write shared memory locations. The order of accesses by those agents is not completely specified in this document. It is possible in some systems for read accesses by I/O devices or other agents to give results indicating some reordering of accesses. However, there are guarantees that apply in all systems. See Section 5.6.4.7.

A shared memory is the primary storage place for one or more locations.

A location is a byte, specified by its physical address. Multiple virtual addresses may map to the same physical address. Ordering considerations are based only on the physical address. This definition of location specifically includes locations and registers in memory mapped I/O devices and bridges to remote I/O (for example, Mailbox Pointer Registers, or MBPRs).

**Implementation Note:**

An implementation may allow a location to have multiple physical addresses, but the rules for accesses via mixtures of the addresses are implementation-specific and outside the scope of this section. Accesses via exactly one of the physical addresses follow the rules described next.

Each processor may generate accesses to shared memory locations. There are six types of accesses:

1. Instruction fetch by processor i to location x, returning value a, denoted $Pi:I<4>(x,a)$.
2. Data read (including load-locked) by processor i to location x, returning value a, denoted $Pi:R<size>(x,a)$.
3. Data write (including successful store-conditional) by processor i to location x, storing value a, denoted $Pi:W<size>(x,a)$.
4. Memory barrier issued by processor i, denoted $Pi:MB$.
5. Write memory barrier issued by processor i, denoted $Pi:WMB$.
6. I-stream memory barrier issued by processor i, denoted $Pi:IMB$. 
The first access type is also called an I-stream access or I-fetch. The next two are also called D-stream accesses. The first three types are collectively called read/write accesses, denoted Pi:Op<\text{}m\text{}angle(x,a), where m is the size of the access in bytes, x is the (physical) address of the access, and a is a value representable in m bytes; for any k in the range 0..m-1, byte k of value a (where byte 0 is the low-order byte) is the value written to or read from location x+k by the access. This relationship reflects little-endian addressing; big-endian addressing representation is as described in Common Architecture, Chapter 2.

The last three types collectively are called barriers or memory barriers.

The size of a read/write access is 8 for a quadword access, 4 for a longword access (including all instruction fetches), 2 for a word access, or 1 for a byte access. All read/write accesses in this chapter are naturally aligned. That is, they have the form Pi:Op<\text{}m\text{}angle(x,a), where the address x is divisible by size m.

The word “access” is also used as a verb; a read/write access Pi:Op<\text{}m\text{}angle(x,a) accesses byte z if x ≤ z < x+m. Two read/write accesses Op1<\text{}m\text{}angle(x,a) and Op2<\text{}n\text{}angle(y,b) are defined to overlap if there is at least one byte that is accessed by both, that is, if max(x,y) < min(x+m,y+n).

5.6.1.1 Architectural Definition of Processor Issue Sequence

The issue sequence for a processor is architecturally defined with respect to a hypothetical simple implementation that contains one processor and a single shared memory, with no caches or buffers. This is the instruction execution model:

1. I-fetch: An Alpha instruction is fetched from memory.
2. Read/Write: That instruction is executed and runs to completion, including a single data read from memory for a Load instruction or a single data write to memory for a Store instruction.
3. Update: The PC for the processor is updated.
4. Loop: Repeat the above sequence indefinitely.

If the instruction fetch step gets a memory management fault, the I-fetch is not done and the PC is updated to point to a PALcode fault handler. If the read/write step gets a memory management fault, the read/write is not done and the PC is updated to point to a PALcode fault handler.

5.6.1.2 Definition of Before and After

The ordering relation BEFORE (⇐) is a partial order on memory accesses. It is further defined in Sections 5.6.1.3 through 5.6.1.9.

The ordering relation BEFORE (⇐), being a partial order, is acyclic.

The BEFORE order cannot be observed directly, nor fully predicted before an actual execution, nor reproduced exactly from one execution to another. Nonetheless, some useful ordering properties must hold in all Alpha implementations.

If u ⇐ v, then v is said to be AFTER u.
5.6.1.3 Definition of Processor Issue Constraints

Processor issue constraints are imposed on the processor issue sequence defined in Section 5.6.1.1.

For two accesses \( u \) and \( v \) issued by processor \( Pi \), if \( u \) precedes \( v \) by processor issue constraint, then \( u \) precedes \( v \) in BEFORE order. \( u \) and \( v \) on \( Pi \) are ordered by processor issue constraint if any of the following applies:

1. The entry in Table 5–1 indicated by the access type of \( u \) (1st) and \( v \) (2nd) indicates the accesses are ordered:

<table>
<thead>
<tr>
<th>1st</th>
<th>2nd</th>
<th>( Pi: &lt;m=4&gt;(x,a) )</th>
<th>( Pi: &lt;n=4&gt;(y,b) )</th>
<th>( Pi: R&lt;n&gt;(y,b) )</th>
<th>( Pi: W&lt;n&gt;(y,b) )</th>
<th>( Pi: MB )</th>
<th>( Pi: IMB )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Pi: &lt;m=4&gt;(x,a) )</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
</tr>
<tr>
<td>( Pi: R&lt;m&gt;(x,a) )</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
</tr>
<tr>
<td>( Pi: W&lt;m&gt;(x,a) )</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
</tr>
<tr>
<td>( Pi: MB )</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
</tr>
<tr>
<td>( Pi: IMB )</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
<td>( \Leftarrow ) if overlap</td>
</tr>
</tbody>
</table>

Where "overlap" denotes the condition \( \max(x,y) < \min(x+m,y+n) \).

2. \( u \) and \( v \) are both writes to memory-like regions and there is a WMB between \( u \) and \( v \) in processor issue sequence.

3. \( u \) and \( v \) are both writes to non-memory-like regions and there is a WMB between \( u \) and \( v \) in processor issue sequence.

4. \( u \) is a TB fill that updates a PTE, for example, a PTE read in order to satisfy a TB miss, and \( v \) is an I- or D-stream access using that PTE (see Sections 5.6.4.3, and 5.6.4.4).

In Table 5–1, 1st and 2nd refer to the ordering of accesses in the processor issue sequence. Note that Table 5–1 imposes no direct constraint on the ordering relationship between nonoverlapping read/write accesses, though there may be indirect constraints due to the transitivity of BEFORE (\( \Leftarrow \)). Conditions 2 through 4, above, impose ordering constraints on some pairs of nonoverlapping read/write accesses.

Table 5–1 permits a read access \( Pi: R<n>(y,b) \) to be ordered BEFORE an overlapping write access \( Pi: W<m>(x,a) \) that precedes the read access in processor issue order. This asymmetry for reads allows reads to be satisfied by using data from an earlier write in processor issue sequence by the same processor (for example, by hitting in a write buffer) before the write completes. The write access remains “visible” to the read access; “visibility” is described in Sections 5.6.1.5 and 5.6.1.6, and illustrated in Litmus Test 11 in Section 5.6.2.11.

An I-fetch \( Pi: <4>(y,b) \) may also be ordered BEFORE an overlapping write \( Pi: W<m>(x,a) \) that precedes it in processor issue sequence. In that case, the write may, but need not, be visible to the I-fetch. This asymmetry in Table 5–1 allows writes to the I-stream to be incoherent until a CALL_PAL IMB is executed.
Implementations are free to perform memory accesses from a single processor in any sequence that is consistent with processor issue constraints.

5.6.1.4 Definition of Location Access Constraints

Location access constraints are imposed on overlapping read/write accesses. If u and v are overlapping read/write accesses, at least one of which is a write, then u and v must be comparable in the BEFORE (\(\leq\)) ordering, that is, either \(u \leq v\) or \(v \leq u\).

There is no direct requirement that nonoverlapping accesses be comparable in the BEFORE (\(\leq\)) ordering.

All writes accessing any given byte are totally ordered, and any read or I-fetch accessing a given byte is ordered with respect to all writes accessing that byte.

5.6.1.5 Definition of Visibility

If u is a write access Pi:W<\textless m\rangle(x,a) and v is an overlapping read access Pj:R<\textless n\rangle(y,b), u is visible to v only if:

\[ u \leq v, \text{ or } u \text{ precedes } v \text{ in processor issue sequence (possible only if } Pi=Pj). \]

If u is a write access Pi:W<\textless m\rangle(x,a) and v is an overlapping instruction fetch Pj:I<\textless 4\rangle(y,b), there are the following rules for visibility:

1. If \(u \leq v\), then u is visible to v.
2. If u precedes v in processor issue sequence, then:
   a. If there is a write w such that:
      \[ u \text{ overlaps } w \text{ and precedes } w \text{ in processor issue sequence, and } w \text{ is visible to } v, \]
      then u is visible to v.
   b. If there is an instruction fetch w such that:
      \[ u \text{ is visible to } w, \text{ and } w \text{ overlaps } v \text{ and precedes } v \text{ in processor issue sequence}, \]
      then u is visible to v.
3. If u does not precede v in either processor issue sequence or BEFORE order, then u is not visible to v.

Note that the rules of visibility for reads and instruction fetches are slightly different. If a write u precedes an overlapping instruction fetch v in processor issue sequence, but u is not BEFORE v, then u may or may not be visible to v.

5.6.1.6 Definition of Storage

The property of storage applies only to memory-like regions.

The value read from any byte by a read access or instruction fetch v, is the value written by the latest (in BEFORE order) write u to that byte, that is visible to v. More formally:
If \( u \) is \( \text{Pi:W}^{<m>}(x,a) \), and \( v \) is either \( \text{Pj:I}^{<4>}(y,b) \) or \( \text{Pj:R}^{<n>}(y,b) \), and \( z \) is a byte accessed by both \( u \) and \( v \), and \( u \) is visible to \( v \); and there is no write that is AFTER \( u \), is visible to \( v \), and accesses byte \( z \); then the value of byte \( z \) read by \( v \) is exactly the value written by \( u \).

The only way to communicate information between different processors is for one to write a shared location and the other to read the shared location and receive the newly written value. (In this context, the sending of an interrupt from processor \( \text{Pi} \) to \( \text{Pj} \) is modeled as \( \text{Pi} \) writing to a location \( \text{INTij} \), and \( \text{Pj} \) reading from \( \text{INTij} \).)

### 5.6.1.7 Definition of Dependence Constraints

The depends order (DP) is defined as follows. Given \( u \) and \( v \) issued by processor \( \text{Pi} \), where \( u \) is a read or an instruction fetch and \( v \) is a write, \( u \) precedes \( v \) in DP order (written \( u \) DP \( v \)), that is, \( v \) depends on \( u \) in either of the following situations:

1. \( u \) determines the execution of \( v \), the location accessed by \( v \), or the value written by \( v \).
2. \( u \) determines the execution or address or value of another memory access \( z \) that precedes \( v \) or might precede \( v \) (that is, would precede \( v \) in some execution path depending on the value read by \( u \)) by processor issue constraint (see Section 5.6.1.3).

### Eliminating Causal Loops

The following rule eliminates the possibility of “causal loops”:

Given that:

\[
\begin{align*}
u &= \text{R0 DP W1} \\
\text{W1 is visible to R1} \\
\text{R1 DP W2} \\
\text{W2 is visible to R2} \\
\text{R2 DP W3} \\
\vdots \\
\text{Wn is visible to Rn} ! \text{ Note that the last W is W0, closing the} \\
\text{Rn DP W0=v} ! \text{ loop to correspond to the first R, R0}
\end{align*}
\]

where:

\( n \geq 1 \), \( \text{Rn} \) is a read or an instruction fetch, \( \text{Wn} \) is a write, and \( \text{Rn,Wn overlap} \), then:

\( u \Leftarrow v \).

A simple example of a “causal loop” is when the execution of a write on \( \text{Pi} \) depends on the execution of a write on \( \text{Pj} \) and vice versa, creating a circular dependence chain. The following simple example of a “causal loop” is written in the style of the litmus tests in Section 5.6.2, where initially \( x \) and \( y \) are 1:

**Processor Pi executes:**

\[
\begin{align*}
\text{LDQ} & \text{ R1, x} \\
\text{STQ} & \text{ R1, y}
\end{align*}
\]

**Processor Pj executes:**
LDQ R1, y  
STQ R1, x

Representing those code sequences in the style of the litmus tests in Section 5.6.2, it is impossible for the following sequence to result:

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1] Pi:R&lt;8&gt;(x,0)</td>
<td>[V1] Pj:R&lt;8&gt;(y,0)</td>
</tr>
<tr>
<td>[U2] Pi:W&lt;8&gt;(y,0)</td>
<td>[V2] Pj:W&lt;8&gt;(x,0)</td>
</tr>
</tbody>
</table>

Analysis:

<1> By the definitions of storage and visibility, U2 is visible to V1, and V2 is visible to U1.

<2> By the definition of DP and examination of the code, U1 DP U2, and V1 DP V2.

<3> Thus U1 DP U2 is visible to V1 DP V2 is visible to U1, which is forbidden by the dependence constraint.

Given the initial condition x,y = 1, the access sequence above would also be impossible if the code were:

Processor Pi’s program:

LDQ R1, x  
BNE R1, done  
STQ R31, y  

done:

Processor Pj’s program:

LDQ R1, y  
BNE R1, done  
STQ R31, x  

done:

5.6.1.8 Definition of Load-Locked and Store-Conditional

The property of load-locked and store-conditional applies only to memory-like regions.

For each successful store-conditional v, there exists a load-locked u such that the following are true:

1. u precedes v in the processor issue sequence.
2. There is no load-locked or store-conditional between u and v in the processor issue sequence.
3. If u and v access within the same naturally aligned 16-byte physical and virtual block in memory, then for every write w by a different processor that accesses
within u’s lock range (where w is either a store or a successful store-conditional),
It must be true that w ⇑ u or v ⇑ w.
u’s lock range contains the region of physical memory that u accesses. See Sections
4.2.4 and 4.2.5, which define the lock range and conditions for success or failure of
a store conditional.

5.6.1.9 Timeliness

Even in the absence of a barrier after the write, no write by a processor may be
delayed indefinitely in the BEFORE ordering.

5.6.2 Litmus Tests

Many issues about writing and reading shared data can be cast into questions
about whether a write is before or after a read. These questions can be answered
by rigorously checking whether any ordering satisfies the rules in Sections 5.6.1.3
through Section 5.6.1.8.

In litmus tests 1–9 below, all initial quadword memory locations contain 1. In all
these litmus tests, it is assumed that initializations are performed by a write or
writes that are BEFORE all the explicitly listed accesses, that all relevant writes
other than the initializations are explicitly shown, and that all accesses shown are
to memory-like regions (so the definition of storage applies).

5.6.2.1 Litmus Test 1 (Impossible Sequence)

\[
\begin{array}{l}
\text{Pi} \\
[U1]\text{Pi:W<8>(x,2)} \\
[V1]\text{Pj:R<8>(x,2)} \\
[V2]\text{Pj:R<8>(x,1)}
\end{array}
\]

Analysis:

<1> By the definition of storage (Section 5.6.1.6), V1 reading 2 implies that U1
is visible to V1.

<2> By the rules for visibility (Section 5.6.1.5), U1 being visible to V1, but being
issued by a different processor, implies that U1 ⇑ V1.

<3> By the processor issue constraints (Section 5.6.1.3), V1 ⇑ V2.

<4> By the transitivity of the partial order ⇑, it follows from <2> and <3> that
U1 ⇑ V2.

<5> By the rules for visibility, it follows from U1 ⇑ V2 that U1 is visible to V2.

<6> Since U1 is AFTER the initialization of x, U1 is the latest (in the ⇑
ordering) write to x that is visible to V1.

<7> By the definition of storage, it follows that V2 should read the value written
by U1, in contradiction to the stated result.

Thus, once a processor reads a new value from a location, it must never see an old
value—time must not go backward. V2 must read 2.
5.6.2.2 Litmus Test 2 (Impossible Sequence)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]Pj:W&lt;8&gt;(x,3)</td>
</tr>
<tr>
<td>[V2]Pj:R&lt;8&gt;(x,2)</td>
<td></td>
</tr>
<tr>
<td>[V3]Pj:R&lt;8&gt;(x,3)</td>
<td></td>
</tr>
</tbody>
</table>

Analysis:
<1> Since V1 precedes V2 in processor issue sequence, V1 is visible to V2.
<2> V2 reading 2 implies U1 is the latest (in \(\leq\) order) write to x visible to V2.
<3> From <1> and <2>, V1 \(\leq\) U1.
<4> Since U1 is visible to V2, and they are issued by different processors, U1 \(\leq\) V2.
<5> By the processor issue constraints, V2 \(\leq\) V3.
<6> From <4> and <5>, U1 \(\leq\) V3.
<7> From <6> and the visibility rules, U1 is visible to V3.
<8> Since both V1 and the initialization of x are BEFORE U1, U1 is the latest write to x that is visible to V3.
<9> By the definition of storage, it follows that V3 should read the value written by U1, in contradiction to the stated result.

Thus, once processor Pj reads a new value written by U1, any other writes that must precede the read must also precede U1. V3 must read 2.

5.6.2.3 Litmus Test 3 (Impossible Sequence)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
<th>Pk</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]Pj:W&lt;8&gt;(x,3)</td>
<td>[W1]Pk:R&lt;8&gt;(x,3)</td>
</tr>
<tr>
<td>[U2]Pi:R&lt;8&gt;(x,3)</td>
<td></td>
<td>[W2]Pk:R&lt;8&gt;(x,2)</td>
</tr>
</tbody>
</table>

Analysis:
<1> U2 reading 3 implies V1 is the latest write to x visible to U2, therefore U1 \(\leq\) V1.
<2> W1 reading 3 implies V1 is visible to W1, so V1 \(\leq\) W1 \(\leq\) W2, therefore V1 is also visible to W2.
<3> W2 reading 2 implies U1 is the latest write to x visible to W2, therefore V1 \(\leq\) U1.
<4> From <1> and <3>, U1 \(\leq\) V1 \(\leq\) U1.

Again, time cannot go backwards. If V1 is ordered before U1, then processor Pk cannot read first the later value 3 and then the earlier value 2. Alternatively, if V1 is ordered before U1, U2 must read 2.
5.6.2.4 Litmus Test 4 (Sequence Okay)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]Pj:R&lt;8&gt;(y,2)</td>
</tr>
<tr>
<td>[U2]Pi:W&lt;8&gt;(y,2)</td>
<td>[V2]Pj:R&lt;8&gt;(x,1)</td>
</tr>
</tbody>
</table>

Analysis:

<1> V1 reading 2 implies U2 ⊑ V1, by storage and visibility.
<2> Since V2 does not read 2, there cannot be U1 ⊑ V2.
<3> By the access order constraints, it follows from <2> that V2 ⊑ U1.

There are no conflicts in the sequence. There are no violations of the definition of BEFORE.

5.6.2.5 Litmus Test 5 (Sequence Okay)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]Pj:R&lt;8&gt;(y,2)</td>
</tr>
<tr>
<td></td>
<td>[V2]Pj:MB</td>
</tr>
<tr>
<td>[U2]Pi:W&lt;8&gt;(y,2)</td>
<td>[V3]Pj:R&lt;8&gt;(x,1)</td>
</tr>
</tbody>
</table>

Analysis:

<1> V1 reading 2 implies U2 ⊑ V1, by storage and visibility.
<2> V1 ⊑ V2 ⊑ V3, by processor issue constraints.
<3> V3 reading 1 implies V3 ⊑ U1, by storage and visibility.

There is U2 ⊑ V1 ⊑ V2 ⊑ V3 ⊑ U1. There are no conflicts in this sequence. There are no violations of the definition of BEFORE.

5.6.2.6 Litmus Test 6 (Sequence Okay)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]Pj:R&lt;8&gt;(y,2)</td>
</tr>
<tr>
<td></td>
<td>[V2]Pj:MB</td>
</tr>
<tr>
<td>[U2]Pi:MB</td>
<td></td>
</tr>
<tr>
<td>[U3]Pi:W&lt;8&gt;(y,2)</td>
<td>[V2]Pj:R&lt;8&gt;(x,1)</td>
</tr>
</tbody>
</table>

Analysis:

<1> U1 ⊑ U2 ⊑ U3, by processor issue constraints.
<2> V1 reading 2 implies U3 ⊑ V1, by storage and visibility.
<3> V2 reading 1 implies V2 ⊑ U1, by storage and visibility.

There is V2 ⊑ U1 ⊑ U2 ⊑ U3 ⊑ V1. There are no conflicts in this sequence. There are no violations of the definition of BEFORE.
In litmus tests 4, 5, and 6, writes to two different locations \( x \) and \( y \) are observed (by another processor) to occur in the opposite order than that in which they were performed. An update to \( y \) propagates quickly to \( P_j \), but the update to \( x \) is delayed, and \( P_i \) and \( P_j \) do not both have MBs.

5.6.2.7 Litmus Test 7 (Impossible Sequence)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]:Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]:Pj:R&lt;8&gt;(y,2)</td>
</tr>
<tr>
<td>[U2]:Pi:MB</td>
<td>[V2]:Pj:MB</td>
</tr>
<tr>
<td>[U3]:Pi:W&lt;8&gt;(y,2)</td>
<td>[V3]:Pj:R&lt;8&gt;(x,1)</td>
</tr>
</tbody>
</table>

Analysis:

1. V3 reading 1 implies \( V_3 \leftarrow U_1 \), by storage and visibility.
2. \( V_1 \) reading 2 implies \( U_3 \leftarrow V_1 \), by storage and visibility.
3. \( U_1 \leftarrow U_2 \leftarrow U_3 \), by processor issue constraints.
4. \( V_1 \leftarrow V_2 \leftarrow V_3 \), by processor issue constraints.
5. By <2>, <3>, and <4>, \( U_1 \leftarrow U_2 \leftarrow U_3 \leftarrow V_1 \leftarrow V_2 \leftarrow V_3 \).

Both <1> and <5> cannot be true, so if \( V_1 \) reads 2, then \( V_3 \) must also read 2.

If both \( x \) and \( y \) are in memory-like regions, the sequence remains impossible if \( U_2 \) is changed to a WMB. Similarly, if both \( x \) and \( y \) are in non-memory-like regions, the sequence remains impossible if \( U_2 \) is changed to a WMB.

5.6.2.8 Litmus Test 8 (Impossible Sequence)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]:Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]:Pj:W&lt;8&gt;(y,2)</td>
</tr>
<tr>
<td>[U2]:Pi:MB</td>
<td>[V2]:Pj:MB</td>
</tr>
<tr>
<td>[U3]:Pi:R&lt;8&gt;(y,1)</td>
<td>[V3]:Pj:R&lt;8&gt;(x,1)</td>
</tr>
</tbody>
</table>

Analysis:

1. V3 reading 1 implies \( V_3 \leftarrow U_1 \), by storage and visibility.
2. \( U_3 \) reading 1 implies \( U_3 \leftarrow V_1 \), by storage and visibility.
3. \( U_1 \leftarrow U_2 \leftarrow U_3 \), by processor issue constraints.
4. \( V_1 \leftarrow V_2 \leftarrow V_3 \), by processor issue constraints.
5. By <2>, <3>, and <4>, \( U_1 \leftarrow U_2 \leftarrow U_3 \leftarrow V_1 \leftarrow V_2 \leftarrow V_3 \).

Both <1> and <5> cannot be true, so if \( U_3 \) reads 1, then \( V_3 \) must read 2, and vice versa.
### 5.6.2.9 Litmus Test 9 (Impossible Sequence)

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;8&gt;(x,2)</td>
<td>[V1]Pj:W&lt;8&gt;(x,3)</td>
</tr>
<tr>
<td>[U2]Pi:R&lt;8&gt;(x,2)</td>
<td>[V2]Pj:R&lt;8&gt;(x,3)</td>
</tr>
<tr>
<td>[U3]Pi:R&lt;8&gt;(x,3)</td>
<td>[V3]Pj:R&lt;8&gt;(x,2)</td>
</tr>
</tbody>
</table>

Analysis:

<1> V3 reading 2 implies U1 is the latest write to x visible to V3, therefore V1 ⇐ U1.

<2> U3 reading 3 implies V1 is the latest write to x visible to U3, therefore U1 ⇐ V1.

Both <1> and <2> cannot be true. Time cannot go backwards. If V3 reads 2, then U3 must read 2. Alternatively, if U3 reads 3, then V3 must read 3.

### 5.6.2.10 Litmus Test 10 (Sequence Okay)

For an aligned quadword location, x, initially 10000000116:

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;4&gt;(x,2)</td>
<td>[V1]Pj:W&lt;4&gt;(x+4,2)</td>
</tr>
<tr>
<td>[U2]Pi:R&lt;8&gt;(x,10000000216)</td>
<td>[V2]Pj:R&lt;8&gt;(x,20000000116)</td>
</tr>
</tbody>
</table>

Analysis:

<1> Since U2 reads 1 from x+4, V1 is not visible to U2. Thus U2 ⇐ V1.

<2> Similarly, V2 ⇐ U1.

<3> U1 is visible to U2, but since they are issued by the same processor, it is not necessarily the case that U1 ⇐ U2.

<4> Similarly, it is not necessarily the case that V1 ⇐ V2.

There is no ordering cycle, so the sequence is permitted.

### 5.6.2.11 Litmus Test 11 (Impossible Sequence)

For an aligned quadword location, x, initially 10000000116:

<table>
<thead>
<tr>
<th>Pi</th>
<th>Pj</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U1]Pi:W&lt;4&gt;(x,2)</td>
<td>[V1]Pj:R&lt;8&gt;(x,20000000116)</td>
</tr>
<tr>
<td>[U2]Pi:MB or WMB</td>
<td></td>
</tr>
<tr>
<td>[U3]Pi:W&lt;4&gt;(x+4,2)</td>
<td></td>
</tr>
</tbody>
</table>

Analysis:
V1 reading 200000001_{16} implies U3 ⇔ V1 ⇔ U1 by storage and visibility.

U1 ⇔ U2 ⇔ U3, by processor issue constraints.

Both <1> and <2> cannot be true.

### 5.6.3 Implied Barriers

There are no implied barriers in Alpha. If an implied barrier is needed for functionally correct access to shared data, it must be written as an explicit instruction. (Software must explicitly include any needed MB, WMB, or CALL_PAL IMB instructions.)

Alpha transitions such as the following have no built-in implied memory barriers:

- Entry to PAL code
- Sending and receiving interrupts
- Returning from exceptions, interrupts, or machine checks
- Swapping context
- Invalidating the Translation Buffer (TB)

Depending on implementation choices for maintaining cache coherency, some PAL code/cache implementations may have an implied CALL_PAL IMB in the I-stream TB fill routine, but this is transparent to the non-PAL code programmer.

### 5.6.4 Implications for Software

Software must explicitly include MB, WMB, or CALL_PAL IMB instructions according to the following circumstances.

#### 5.6.4.1 Single-Processor Data Stream

No barriers are ever needed. A read to physical address $x$ will always return the value written by the immediately preceding write to $x$ in the processor issue sequence.

#### 5.6.4.2 Single-Processor Instruction Stream

An I-fetch from virtual or physical address $x$ does not necessarily return the value written by the immediately preceding write to $x$ in the issue sequence. To make the I-fetch reliably get the newly written instruction, a CALL_PAL IMB is needed between the write and the I-fetch.
5.6.4.3 Multiple-Processor Data Stream (Including Single Processor with DMA I/O)

Generally, the only way to reliably communicate shared data is to write the shared data on one processor or DMA I/O device, execute an MB (or the logical equivalent\(^1\) if it is a DMA I/O device), then write a flag (equivalently, send an interrupt) signaling the other processor that the shared data is ready. Each receiving processor must read the new flag (equivalently, receive the interrupt), execute an MB, then read or update the shared data. In the special case in which data is communicated through just one location in memory, memory barriers are not necessary.

Software Note:

Note that this section does not describe how to reliably communicate data from a processor to a DMA device. See Section 5.6.4.7.

Leaving out the first MB removes the assurance that the shared data is written before the flag is written.

Leaving out the second MB removes the assurance that the shared data is read or updated only after the flag is seen to change; in this case, an early read could see an old value, and an early update could be overwritten.

This implies that after a DMA I/O device has written some data to memory (such as paging in a page from disk), the DMA device must logically execute an MB\(^1\) before posting a completion interrupt, and the interrupt handler software must execute an MB before the data is guaranteed to be visible to the interrupted processor. Other processors must also execute MBs before they are guaranteed to see the new data.

An important special case occurs when a write is done (perhaps by an I/O device) to some physical page frame, then an MB is executed, and then a previously invalid PTE is changed to be a valid mapping of the physical page frame that was just written. In this case, all processors that access virtual memory by using the newly valid PTE must guarantee to deliver the newly written data after the TB miss, for both I-stream and D-stream accesses.

5.6.4.4 Multiple-Processor Instruction Stream (Including Single Processor with DMA I/O)

The only way to update the I-stream reliably is to write the shared I-stream on one processor or DMA I/O device, then execute a CALL_PAL IMB (or an MB if the processor is not going to execute the new I-stream, or the logical equivalent of an MB if it is a DMA I/O device), then write a flag (equivalently, send an interrupt) signaling the other processor that the shared I-stream is ready. Each receiving processor must read the new flag (equivalently, receive the interrupt), execute a CALL_PAL IMB, then fetch the shared I-stream.

\(^1\) In this context, the logical equivalent of an MB for a DMA device is whatever is necessary under the applicable I/O subsystem architecture to ensure that preceding writes will be BEFORE (see Section 5.6.1.2) the subsequent write of a flag or transmission of an interrupt. Not all I/O devices behave exactly as required by the Alpha architecture. To interoperate properly with those devices, some special action might be required by the program executing on the CPU. For example, PCI bus devices require that after the CPU has received an interrupt, the CPU must read a CSR location on the PCI device, execute an MB, then read or update the shared data. From the perspective of the Alpha architecture, this CSR read can be regarded as a necessary assist to help the DMA I/O device complete its logical equivalent of an MB.
Software Note:

Note that this section does not describe how to reliably communicate I-stream from a processor to a DMA device. See Section 5.6.4.7.

Leaving out the first CALL_PAL IMB (or MB) removes the assurance that the shared I-stream is written before the flag.

Leaving out the second CALL_PAL IMB removes the assurance that the shared I-stream is read only after the flag is seen to change; in this case, an early read could see an old value.

This implies that after a DMA I/O device has written some I-stream to memory (such as paging in a page from disk), the DMA device must logically execute an MB before posting a completion interrupt, and the interrupt handler software must execute a CALL_PAL IMB before the I-stream is guaranteed to be visible to the interrupted processor. Other processors must also execute CALL_PAL IMB instructions before they are guaranteed to see the new I-stream.

An important special case occurs under the following circumstances:

1. A write (perhaps by an I/O device) is done to some physical page frame.
2. A CALL_PAL IMB (or MB) is executed.
3. A previously invalid PTE is changed to be a valid mapping of the physical page frame that was written in step 1.

In this case, all processors that access virtual memory by using the newly valid PTE must guarantee to deliver the newly written I-stream after the TB miss.

5.6.4.5 Multiple-Processor Context Switch

If a process migrates from executing on one processor to executing on another, the context switch operating system code must include a number of barriers.

A process migrates by having its context stored into memory, then eventually having that context reloaded on another processor. In between, some shared mechanism must be used to communicate that the context saved in memory by the first processor is available to the second processor. This could be done by using an interrupt, by using a flag bit associated with the saved context, or by using a shared-memory multiprocessor data structure, as follows:

<table>
<thead>
<tr>
<th>First Processor</th>
<th>Second Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save state of current process. MB [1]</td>
<td>Pick up ownership of process context data structure memory. MB [2]</td>
</tr>
<tr>
<td>Pass ownership of process context data structure memory.</td>
<td>⇒</td>
</tr>
<tr>
<td>First Processor</td>
<td>Second Processor</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Restore state of new process context data structure memory. Make I-stream coherent [3]. Make TB coherent [4].</td>
<td>Execute code for new process that accesses memory that is not common to all processes.</td>
</tr>
</tbody>
</table>

MB [1] ensures that the writes done to save the state of the current process happen before the ownership is passed.

MB [2] ensures that the reads done to load the state of the new process happen after the ownership is picked up and hence are reliably the values written by the processor saving the old state. Leaving this MB out makes the code fail if an old value of the context remains in the second processor’s cache and invalidates from the writes done on the first processor are not delivered soon enough.

The TB on the second processor must be made coherent with any write to the page tables that may have occurred on the first processor just before the save of the process state. This must be done with a series of TB invalidate instructions to remove any nonglobal page mapping for this process, or by assigning an ASN that is unused on the second processor to the process. One of these actions must occur sometime before starting execution of the code for the new process that accesses memory (instruction or data) that is not common to all processes. A common method is to assign a new ASN after gaining ownership of the new process and before loading its context, which includes its ASN.

The D-cache on the second processor must be made coherent with any write to the D-stream that may have occurred on the first processor just before the save of process state. This is ensured by MB [2] and does not require any additional instructions.

The I-cache on the second processor must be made coherent with any write to the I-stream that may have occurred on the first processor just before the save of process state. This can be done with a CALL_PAL IMB sometime before the execution of any code that is not common to all processes. More commonly, this can be done by forcing a TB miss (via the new ASN or via TB invalidate instructions) and using the TB-fill rule (see Section 5.6.4.3). This latter approach does not require any additional instruction.
Combining all these considerations gives:

<table>
<thead>
<tr>
<th>First Processor</th>
<th>Second Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Pick up ownership of process context data structure memory.</td>
<td>Pickup ownership of new process context data structure memory.</td>
</tr>
<tr>
<td>MB</td>
<td>MB</td>
</tr>
<tr>
<td>Assign new ASN or invalidate TBs.</td>
<td>Assign new ASN or invalidate TBs.</td>
</tr>
<tr>
<td>Save state of current process.</td>
<td>Save state of current process.</td>
</tr>
<tr>
<td>Restore state of new process.</td>
<td>Restore state of new process.</td>
</tr>
<tr>
<td>MB</td>
<td>MB</td>
</tr>
<tr>
<td>Pass ownership of process context data structure memory. ⇒</td>
<td>Pass ownership of old process context data structure memory. ⇒</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

On a single processor, there is no need for the barriers.

5.6.4.6 Multiple-Processor Send/Receive Interrupt

If one processor writes some shared data, then sends an interrupt to a second processor, and that processor receives the interrupt, then accesses the shared data, the sequence from Section 5.6.4.3 must be used:

<table>
<thead>
<tr>
<th>First Processor</th>
<th>Second Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Write data</td>
<td>Receive interrupt</td>
</tr>
<tr>
<td>MB</td>
<td>MB</td>
</tr>
<tr>
<td>Send interrupt ⇒</td>
<td>Access data</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
</tbody>
</table>

Leaving out the MB at the beginning of the interrupt-receipt routine causes the code to fail if an old value of the context remains in the second processor’s cache.
and invalidates from the writes done on the first processor are not delivered soon enough.

5.6.4.7 Implications for Memory Mapped I/O

Sections 5.6.4.3 and 5.6.4.4 describe methods for communicating data from a processor or DMA I/O device to another processor that work reliably in all Alpha systems. Special considerations apply to the communication of data or I-stream from a processor to a DMA I/O device. These considerations arise from the use of bridges to connect to I/O buses with devices that are accessible by memory accesses to non-memory-like regions of physical memory.

The following communication method works in all Alpha systems.

To reliably communicate shared data from a processor to an I/O device:

1. Write the shared data to a memory-like physical memory region on the processor.
2. Execute an MB instruction.
3. Write a flag (equivalently, send an interrupt or write a register location implemented in the I/O device).

The receiving I/O device must:

1. Read the flag (equivalently, detect the interrupt or detect the write to the register location implemented in the I/O device).
2. Execute the equivalent of an MB\(^1\)
3. Read the shared data.

As shown in Section 5.6.4.3, leaving out the memory barrier removes the assurance that the shared data is written before the flag is. Unlike the case in Section 5.6.4.3, writing the shared data to a non-memory-like physical memory region removes the assurance that the I/O device will detect the writes of the shared data before detecting the flag write, interrupt, or device register write.

This implies that after a processor has prepared a data buffer to be read from memory by a DMA I/O device (such as writing a buffer to disk), the processor must execute an MB before starting the I/O. The I/O device, after receiving the start signal, must logically execute an MB before reading the data buffer, and the buffer must be located in a memory-like physical memory region.

There are methods of communicating data that may work in some systems but are not guaranteed in all systems. Two notable examples are:

1. If an Alpha processor writes a location implemented in a component located on an I/O bus in the system, then executes a memory barrier, then writes a flag in some memory location (in a memory-like or non-memory-like region), a device

\(^1\) In this context, the logical equivalent of an MB for a DMA device is whatever is necessary under the applicable I/O subsystem architecture to ensure that preceding writes will be BEFORE (see Section 5.6.1.2) the subsequent reads of shared data. Typically, this action is defined to be present between every read and write access done by the I/O device, according to the applicable I/O subsystem architecture.
on the I/O bus may be able to detect (via read access) the result of the flag in memory write and the write of the location on the I/O bus out of order (that is, in a different order than the order in which the Alpha processor wrote those locations).

2. If an Alpha processor writes a location that is a control register within an I/O device, then executes a memory barrier, then writes a location in memory (in a memory-like or non-memory-like region), the I/O device may be able to detect (via read access) the result of the memory write before receiving and responding to the write of its own control register.

In almost every case, a mechanism that ensures the completion of writes to control register locations within I/O devices is provided. The normal and strongly recommended mechanism is to read a location after writing it, which guarantees that the write is complete. In any case, all systems that use a particular I/O device should provide the same mechanism for that device.

5.6.4.8 Multiple Processors Writing to a Single I/O Device

Generally, for multiple processors to cooperate in writing to a single I/O device, the first processor must write to the device, execute an MB, then somehow notify other processors. Another processor that intends to write the same I/O device after the first processor must receive the notification, execute an MB, and then write to the I/O device.

For example:

<table>
<thead>
<tr>
<th>First Processor</th>
<th>Second Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write CSR_A</td>
<td></td>
</tr>
<tr>
<td>MB</td>
<td>Read flag (in memory)</td>
</tr>
<tr>
<td>Write flag (in memory)</td>
<td>Execute an MB</td>
</tr>
<tr>
<td></td>
<td>Write CSR_B</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The MB on the first processor guarantees that the write to CSR_A precedes the write to flag in memory, as perceived on other processors. (The MB does not guarantee that the write to CSR_A has completed. See Section 5.6.4.7 for a discussion of how a processor can guarantee that a write to an I/O device has completed at that device.) The MB on the second processor guarantees that the write to CSR_B will reach the I/O device after the write to CSR_A.

5.6.5 Implications for Hardware

The coherency point for physical address x is the place in the memory subsystem at which accesses to x are ordered. It may be at a main memory board, or at a cache containing x exclusively, or at the point of winning a common bus arbitration.
The coherency point for x may move with time, as exclusive access to x migrates between main memory and various caches.

MB and CALL_PAL IMB force all preceding writes to at least reach their respective coherency points. This does not mean that main-memory writes have been done, just that the order of the eventual writes is committed. For example, on the XMI with retry, this means getting the writes acknowledged as received with good parity at the inputs to memory board queues; the actual RAM write happens later.

MB and CALL_PAL IMB also force all queued cache invalidates to be delivered to the local caches before starting any subsequent reads (that may otherwise cache hit on stale data) or writes (that may otherwise write the cache, only to have the write effectively overwritten by a late-delivered invalidate).

WMB ensures that the final order of writes to memory-like regions is committed and that the final order of writes to non-memory-like regions is committed. This does not imply that the final order of writes to memory-like regions relative to writes to non-memory-like regions is committed. It also prevents writes that precede the WMB from merging with writes that follow the WMB. For example, an implementation with a write buffer might implement WMB by closing all valid write buffer entries from further merging and then drain the write buffer entries in order.

Implementations may allow reads of x to hit (by physical address) on pending writes in a write buffer, even before the writes to x reach the coherency point for x. If this is done, it is still true that no earlier value of x may subsequently be delivered to the processor that took the hit on the write buffer value.

Virtual data caches are allowed to deliver data before doing address translation, but only if there cannot be a pending write under a synonym virtual address. Lack of a write-buffer match on untranslated address bits is sufficient to guarantee this.

Virtual data caches must invalidate or otherwise become coherent with the new value whenever a PALcode routine is executed that affects the validity, fault behavior, protection behavior, or virtual-to-physical mapping specified for one or more pages. Becoming coherent can be delayed until the next subsequent MB instruction or TB fill (using the new mapping) if the implementation of the PALcode routine always forces a subsequent TB fill.

5.7 Arithmetic Traps

Alpha implementations are allowed to execute multiple instructions concurrently and to forward results from one instruction to another. Thus, when an arithmetic trap is detected, the PC may have advanced an arbitrarily large number of instructions past the instruction T (calculating result R) whose execution triggered the trap.

When the trap is detected, any or all of these subsequent instructions may run to completion before the trap is actually taken. Instruction T and the set of instructions subsequent to T that complete before the trap is taken are collectively called the trap shadow of T. The PC pushed on the stack when the trap is taken is the PC of the first instruction past the trap shadow.
The instructions in the trap shadow of T may use the UNPREDICTABLE result R of T, they may generate additional traps, and they may completely change the PC (branches, JSR).

Thus, by the time a trap is taken, the PC pushed on the stack may bear no useful relationship to the PC of the trigger instruction T, and the state visible to the programmer may have been updated using the UNPREDICTABLE result R. If an instruction in the trap shadow of T uses R to calculate a subsequent register value, that register value is UNPREDICTABLE, even though there may be no trap associated with the subsequent calculation. Similarly:

- If an instruction in the trap shadow of T stores R or any subsequent UNPREDICTABLE result, the stored value is UNPREDICTABLE.
- If an instruction in the trap shadow of T uses R or any subsequent UNPREDICTABLE result as the basis of a conditional or calculated branch, the branch target is UNPREDICTABLE.
- If an instruction in the trap shadow of T uses R or any subsequent UNPREDICTABLE result as the basis of an address calculation, the memory address actually accessed is UNPREDICTABLE.

Software that is intended to bound how far the PC may advance before taking a trap, or how far an UNPREDICTABLE result may propagate, must insert TRAPB instructions at appropriate points.

Software that is intended to continue from a trap by supplying a well-defined result R within an arithmetic trap handler, can do so reliably by following the rules for software completion code sequences given in Section 4.7.6.
6.1 PALcode

In a family of machines, both users and operating system implementors require functions to be implemented consistently. When functions conform to a common interface, the code that uses those functions can be used on several different implementations without modification.

These functions range from the binary encoding of the instruction and data to the exception mechanisms and synchronization primitives. Some of these functions can be implemented cost effectively in hardware, but others are impractical to implement directly in hardware. These functions include low-level hardware support functions such as Translation Buffer miss fill routines, interrupt acknowledge, and vector dispatch. They also include support for privileged and atomic operations that require long instruction sequences.

In the VAX, these functions are generally provided by microcode. This is not seen as a problem because the VAX architecture lends itself to a microcoded implementation.

One of the goals of Alpha architecture is to implement functions consistently without microcode. However, it is still desirable to provide an architected interface to these functions that will be consistent across the entire family of machines. The Privileged Architecture Library (PALcode) provides a mechanism to implement these functions without microcode.

6.2 PALcode Instructions and Functions

PALcode is used to implement the following functions:

• Instructions that require complex sequencing as an atomic operation
• Instructions that require VAX style interlocked memory access
• Privileged instructions
• Memory management control, including translation buffer (TB) management
• Context swapping
• Interrupt and exception dispatching
• Power-up initialization and booting
• Console functions
• Emulation of instructions with no hardware support
The Alpha architecture lets these functions be implemented in standard machine code that is resident in main memory. PALcode is written in standard machine code with some implementation-specific extensions to provide access to low-level hardware. This lets an Alpha implementation make various design trade-offs based on the hardware technology being used to implement the machine. The PALcode can abstract these differences and make them invisible to system software.

For example, in a MOS VLSI implementation, a small (32-entry) fully associative TB can be the right match to the media, given that chip area is a costly resource. In an ECL version, a large (1024 entry) direct-mapped TB can be used because it will use RAM chips and does not have fast associative memories available. This difference would be handled by implementation-specific versions of the PALcode on the two systems, both versions providing transparent TB miss service routines. The operating system code would not need to know there were any differences.

An Alpha Privileged Architecture Library (PALcode) of routines and environments is supplied by Digital. Other systems may use a library supplied by Digital or architect and implement a different library of routines. Alpha systems are required to support the replacement of PALcode defined by Digital with an operating system-specific version.

6.3 PALcode Environment

The PALcode environment differs from the normal environment in the following ways:

- Complete control of the machine state.
- Interrupts are disabled.
- Implementation-specific hardware functions are enabled, as described below.
- I-stream memory management traps are prevented (by disabling I-stream mapping, mapping PALcode with a permanent TB entry, or by other mechanisms).

Complete control of the machine state allows all functions of the machine to be controlled. Disabling interrupts allows the system to provide multi-instruction sequences as atomic operations. Enabling implementation-specific hardware functions allows access to low-level system hardware. Preventing I-stream memory management traps allows PALcode to implement memory management functions such as translation buffer fill.

6.4 Special Functions Required for PALcode

PALcode uses the Alpha instruction set for most of its operations. A small number of additional functions are needed to implement the PALcode. Five opcodes are reserved to implement PALcode functions: PAL19, PAL1B, PAL1D, PAL1E, and PAL1F. These instructions produce an trap if executed outside the PALcode environment.
• PALcode needs a mechanism to save the current state of the machine and dispatch into PALcode.

• PALcode needs a set of instructions to access hardware control registers.

• PALcode needs a hardware mechanism to transition the machine from the PALcode environment to the non-PALcode environment. This mechanism loads the PC, enables interrupts, enables mapping, and disables PALcode privileges.

An Alpha implementation may also choose to provide additional functions to simplify or improve performance of some PALcode functions. The following are some examples:

• An Alpha implementation may include a read/write virtual function that allows PALcode to perform mapped memory accesses using the mapping hardware rather than providing the virtual-to-physical translation in PALcode routines. PALcode may provide a special function to do physical reads and writes and have the Alpha loads and stores continue to operate on virtual address in the PALcode environment.

• An Alpha implementation may include hardware assists for various functions, such as saving the virtual address of a reference on a memory management error rather than having to generate it by simulating the effective address calculation in PALcode.

• An Alpha implementation may include private registers so it can function without having to save and restore the native general registers.

6.5 PALcode Effects on System Code

PALcode will have one effect on system code. Because PALcode may reside in main memory and maintain privileged data structures in main memory, the operating system code that allocates physical memory cannot use all of physical memory.

The amount of memory PALcode requires is small, so the loss to the system is negligible.

6.6 PALcode Replacement

Alpha systems are required to support the replacement of PALcode supplied by Digital with an operating system-specific version. The following functions must be implemented in PALcode, not directly in hardware, to facilitate replacement with different versions.

1. Translation Buffer fill. Different operating systems will want to replace the Translation Buffer (TB) fill routines. The replacement routines will use different data structures. Page tables will not be present in these systems. Therefore, no portion of the TB fill flow that would change with a change in page tables may be placed in hardware, unless it is placed in a manner that can be overridden by PALcode.
2. Process structure. Different operating systems might want to replace the process context switch routines. The replacement routines will use different data structures. The HWPCB or PCB will not be present in these systems. Therefore, no portion of the context switching flows that would change with a change in process structure may be placed in hardware.

PALcode can be viewed as consisting of the following somewhat intertwined components:

- Chip/architecture component
- Hardware platform component
- Operating system component

PALcode should be written modularly to facilitate the easy replacement or conditional building of each component. Such a practice simplifies the integration of CPU hardware, system platform hardware, console firmware, operating system software, and compilers.

PALcode subsections that are commonly subject to modification include:

- Translation Buffer fill
- Process structure and context switch
- Interrupt and exception frame format and routine dispatch
- Privileged PALcode instructions
- Transitions to and from console I/O mode
- Power-up reset

6.7 Required PALcode Instructions

The PALcode instructions listed in Table 6-1 and Appendix C must be recognized by mnemonic and opcode in all operating system implementations, but the effect of each instruction is dependent on the implementation. Digital defines the operation of these PALcode instructions for operating system implementations supplied by Digital.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPT</td>
<td>Breakpoint trap</td>
</tr>
<tr>
<td>BUGCHK</td>
<td>Bugcheck trap</td>
</tr>
<tr>
<td>CSERVE</td>
<td>Console service</td>
</tr>
<tr>
<td>GENTRAP</td>
<td>Generate trap</td>
</tr>
<tr>
<td>RDUNIQUE</td>
<td>Read unique value</td>
</tr>
</tbody>
</table>
Table 6–1 (Cont.): PALcode Instructions that Require Recognition

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWPPAL</td>
<td>Swap PAL code</td>
</tr>
<tr>
<td>WRUNIQUE</td>
<td>Write unique value</td>
</tr>
</tbody>
</table>

The PAL code instructions listed in Table 6–2 and described in the following sections must be supported by all Alpha implementations:

Table 6–2: Required PALcode Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Type</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAIN A</td>
<td>Privileged</td>
<td>Drain aborts</td>
</tr>
<tr>
<td>HALT</td>
<td>Privileged</td>
<td>Halt processor</td>
</tr>
<tr>
<td>IMB</td>
<td>Unprivileged</td>
<td>1-stream memory barrier</td>
</tr>
</tbody>
</table>
6.7.1 Drain Aborts

Format:

```
CALL_PAL  DRAINA
```

Operation:

```
IF  PS<CM> NE 0 THEN
    {privileged instruction exception}
    (Stall instruction issuing until all prior
    instructions are guaranteed to complete
    without incurring aborts.)
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

```
CALL_PAL  DRAINA
```

Description:

If aborts are deliberately generated and handled (such as nonexistent memory aborts while sizing memory or searching for I/O devices), the DRAINA instruction forces any outstanding aborts to be taken before continuing.

Aborts are necessarily implementation dependent. DRAINA stalls instruction issue at least until all previously issued instructions have completed and any associated aborts have been signaled, as follows:

- For operate instructions, this usually means stalling until the result register has been written.
- For branch instructions, this usually means stalling until the result register and PC have been written.
- For load instructions, this usually means stalling until the result register has been written.
- For store instructions, this usually means stalling until at least the first level in a potentially multilevel memory hierarchy has been written.

For load instructions, DRAINA does not necessarily guarantee that the unaccessed portions of a cache block have been transferred error free before continuing.
For store instructions, DRAINNA does not necessarily guarantee that the ultimate target location of the store has received error-free data before continuing. An implementation-specific technique must be used to guarantee the ultimate completion of a write in implementations that have multilevel memory hierarchies or store-and-forward bus adapters.
6.7.2 Halt

Format:

```
CALL_PAL   HALT

!PAL code format
```

Operation:

```
IF  PS<CM>  NE  0  THEN
   {privileged instruction exception}
CASE {halt_action} OF
   ! Operating System or Platform dependent choice
   halt: {halt}
   restart/boot/halt: {restart/boot/halt}
   boot/halt: {boot/halt}
   debugger/halt: {debugger/halt}
   restart/halt: {restart/halt}
ENDCASE
```

Exceptions:

Privileged Instruction

Instruction mnemonics:

```
CALL_PAL HALT    Halt Processor
```

Description:

The HALT instruction stops normal instruction processing and initiates some other
operating system or platform-specific behavior, depending on the HALT action
setting. The choice of behavior typically includes the initiation of a restart sequence,
a system bootstrap, or entry into console mode. See Console Interface (III), Chapter
3.
6.7.3 Instruction Memory Barrier

Format:

CALL_PAL IMB

PAL code format

Operation:

{Make instruction stream coherent with Data stream}

Exceptions:

None

Instruction mnemonics:

CALL_PAL IMB I-stream Memory Barrier

Description:

An IMB instruction must be executed after software or I/O devices write into the instruction stream or modify the instruction stream virtual address mapping, and before the new value is fetched as an instruction. An implementation may contain an instruction cache that does not track either processor or I/O writes into the instruction stream. The instruction cache and memory are made coherent by an IMB instruction.

If the instruction stream is modified and an IMB is not executed before fetching an instruction from the modified location, it is UNPREDICTABLE whether the old or new value is fetched.

Software Note:

In a multiprocessor environment, executing an IMB on one processor does not affect instruction caches on other processors. Thus, a single IMB on one processor is insufficient to guarantee that all processors see a modification of the instruction stream.

The cache coherency and sharing rules are described in Chapter 5.
On an Alpha system, underlying control of the system platform hardware is provided by a console subsystem. The console subsystem:

1. Initializes, tests, and prepares the system platform hardware for Alpha system software.
2. Bootstraps (loads into memory and starts the execution of) system software.
3. Controls and monitors the state and state transitions of each processor in a multiprocessor system.
4. Provides services to system software that simplify system software control of and access to platform hardware.
5. Provides a means for a console operator to monitor and control the system.

The console subsystem interacts with system platform hardware to accomplish the first three tasks. The actual mechanisms of these interactions are specific to the platform hardware; however, the net effects are common to all systems.

The console subsystem interacts with system software once control of the system platform hardware has been transferred to that software.

The console subsystem interacts with the console operator through a virtual display device or console terminal. The console operator may be a person or a management application.
Chapter 8

Input/Output Overview (I)

Conceptually, Alpha systems can consist of processors, memory, a processor-memory interconnect (PMI), I/O buses, bridges, and I/O devices.

Figure 8–1 shows the Alpha system overview.

**Figure 8–1: Alpha System Overview**

As shown in Figure 8–1, processors, memory, and possibly I/O devices, are connected by a PMI.

A bridge connects an I/O bus to the system, either directly to the PMI or through another I/O bus. The I/O bus address space is available to the processor either directly or indirectly. Indirect access is provided through either an I/O mailbox or an I/O mapping mechanism. The I/O mapping mechanism includes provisions for mapping between PMI and I/O bus addresses and access to I/O bus operations.

Alpha I/O operations can include:

- Accesses between the processor and an I/O device across the PMI
- Accesses between the processor and an I/O device across an I/O bus
- DMA accesses — I/O devices initiating reads and writes to memory
- Processor interrupts requested by devices
- Bus-specific I/O accesses
The following sections specify the Privileged Architecture Library (PALcode) instructions, that are required to support an OpenVMS Alpha system.

9.1 Unprivileged OpenVMS Alpha PALcode Instructions

The unprivileged PAL code instructions provide support for system operations to all modes of operation (kernel, executive, supervisor, and user).

Table 9–1 describes the unprivileged OpenVMS Alpha PALcode instructions.

Table 9–1: Unprivileged OpenVMS Alpha PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPT</td>
<td>Breakpoint</td>
</tr>
<tr>
<td></td>
<td>The BPT instruction is provided for program debugging. It switches the processor to kernel mode and pushes R2..R7, the updated PC, and PS on the kernel stack. It then dispatches to the address in the Breakpoint vector, stored in a control block.</td>
</tr>
<tr>
<td>BUGCHK</td>
<td>Bugcheck</td>
</tr>
<tr>
<td></td>
<td>The BUGCHK instruction is provided for error reporting. It switches the processor to kernel mode and pushes R2..R7, the updated PC, and PS on the kernel stack. It then dispatches to the address in the bugcheck vector, stored in a control block.</td>
</tr>
<tr>
<td>CHME</td>
<td>Change mode to executive</td>
</tr>
<tr>
<td></td>
<td>The CHME instruction allows a process to change its mode in a controlled manner. A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. Registers R2..R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHME instruction.</td>
</tr>
<tr>
<td>CHMK</td>
<td>Change mode to kernel</td>
</tr>
<tr>
<td></td>
<td>The CHMK instruction allows a process to change its mode to kernel in a controlled manner. A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PS, and PC are pushed onto the kernel stack. The saved PC addresses the instruction following the CHMK instruction.</td>
</tr>
</tbody>
</table>
Table 9–1 (Cont.): Unprivileged OpenVMS Alpha PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHMS</td>
<td>Change mode to supervisor</td>
</tr>
<tr>
<td></td>
<td>The CHMS instruction allows a process to change its mode in a controlled manner.</td>
</tr>
<tr>
<td></td>
<td>A change in mode also results in a change of stack pointers: the old pointer is saved, the new pointer is loaded. R2..R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction.</td>
</tr>
<tr>
<td>CHMU</td>
<td>Change mode to user</td>
</tr>
<tr>
<td></td>
<td>The CHMU instruction allows a process to call a routine via the change mode mechanism.</td>
</tr>
<tr>
<td></td>
<td>R2..R7, PS, and PC are pushed onto the current stack. The saved PC addresses the instruction following the CHMU instruction.</td>
</tr>
<tr>
<td>CLRFEN</td>
<td>Clear floating-point enable</td>
</tr>
<tr>
<td></td>
<td>The CLRFEN instruction writes a zero to the floating-point enable register.</td>
</tr>
<tr>
<td>GENTRAP</td>
<td>Generate trap</td>
</tr>
<tr>
<td></td>
<td>The GENTRAP instruction is provided for reporting runtime software conditions. It switches the processor to kernel mode and pushes registers R2..R7, the updated PC, and the PS on the kernel stack. It then dispatches to the address of the GENTRAP vector, stored in a control block.</td>
</tr>
<tr>
<td>IMB</td>
<td>I-Stream memory barrier</td>
</tr>
<tr>
<td></td>
<td>The IMB instruction ensures that the contents of an instruction cache are coherent after the instruction stream has been modified by software or I/O devices.</td>
</tr>
<tr>
<td></td>
<td>If the instruction stream is modified and an IMB is not executed before fetching an instruction from the modified location, it is UNPREDICTABLE whether the old or new value is fetched.</td>
</tr>
<tr>
<td>INSQHIL</td>
<td>Insert into longword queue at header, interlocked</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td>INSQHILR</td>
<td>Insert into longword queue at header, interlocked resident</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td></td>
<td>This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation and Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INSQHIQ</td>
<td>Insert into quadword queue at header, interlocked</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td>INSQHIQR</td>
<td>Insert into quadword queue at header, interlocked resident</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue following the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td></td>
<td>This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.</td>
</tr>
<tr>
<td>INSQTIL</td>
<td>Insert into longword queue at tail, interlocked</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td>INSQTILR</td>
<td>Insert into longword queue at tail, interlocked resident</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td></td>
<td>This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.</td>
</tr>
<tr>
<td>INSQTIQ</td>
<td>Insert into quadword queue at tail, interlocked</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td>INSQTIQR</td>
<td>Insert into quadword queue at tail, interlocked resident</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the self-relative queue preceding the header specified in R16. The insertion is a noninterruptible operation. The insertion is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment.</td>
</tr>
<tr>
<td></td>
<td>This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.</td>
</tr>
</tbody>
</table>
### Table 9–1 (Cont.): Unprivileged OpenVMS Alpha PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSQUEL</td>
<td>Insert into longword queue</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the absolute queue following the entry specified by the predecessor addressed by R16 for INSQUEL, or following the entry specified by the contents of the longword addressed by R16 for INSQUEL/D. The insertion is a noninterruptible operation.</td>
</tr>
<tr>
<td>INSQUEQ</td>
<td>Insert into quadword queue</td>
</tr>
<tr>
<td></td>
<td>The entry specified in R17 is inserted into the absolute queue following the entry specified by the predecessor addressed by R16 for INSQUEQ, or following the entry specified by the contents of the quadword addressed by R16 for INSQUEQ/D. The insertion is a noninterruptible operation.</td>
</tr>
<tr>
<td>PROBE</td>
<td>Probe read/write access</td>
</tr>
<tr>
<td></td>
<td>PROBE checks the read (PROBER) or write (PROBEW) accessibility of the first and last byte specified by the base address and the signed offset; the bytes in between are not checked. System software must check all pages between the two bytes if they are to be accessed.</td>
</tr>
<tr>
<td></td>
<td>PROBE is only intended to check a single datum for accessibility.</td>
</tr>
<tr>
<td>RD_PS</td>
<td>Read processor status</td>
</tr>
<tr>
<td></td>
<td>RD_PS writes the Processor Status (PS) to register R0.</td>
</tr>
<tr>
<td>READ_UNQ</td>
<td>Read unique context</td>
</tr>
<tr>
<td></td>
<td>READ_UNQ reads the hardware process (thread) unique context value, if previously written by WRITE_UNQ, and places that value in R0.</td>
</tr>
<tr>
<td>REI</td>
<td>Return from exception or interrupt</td>
</tr>
<tr>
<td></td>
<td>The PS, PC, and saved R2..R7 are popped from the current stack and held in temporary registers. The new PS is checked for validity and consistency. If it is valid and consistent, the current stack pointer is then saved and a new stack pointer is selected. Registers R2 through R7 are restored by using the saved values held in the temporary registers. A check is made to determine if an AST or interrupt is pending.</td>
</tr>
<tr>
<td></td>
<td>If the enabling conditions are present for an interrupt or AST at the completion of this instruction, the interrupt or AST occurs before the next instruction.</td>
</tr>
<tr>
<td>REMQHIL</td>
<td>Remove from longword queue at header, interlocked</td>
</tr>
<tr>
<td></td>
<td>The self-relative queue entry following the header, pointed to by R16, is removed from the queue, and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation and Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>REMQHILR</td>
<td>Remove from longword queue at header, interlocked resident</td>
</tr>
<tr>
<td></td>
<td>The queue entry following the header, pointed to by R16, is removed from the self-relative queue, and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.</td>
</tr>
<tr>
<td>REMQHIQ</td>
<td>Remove from quadword queue at header, interlocked</td>
</tr>
<tr>
<td></td>
<td>The self-relative queue entry following the header, pointed to by R16, is removed from the queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.</td>
</tr>
<tr>
<td>REMQHIQR</td>
<td>Remove from quadword queue at header, interlocked resident</td>
</tr>
<tr>
<td></td>
<td>The queue entry following the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned.</td>
</tr>
<tr>
<td>REMQTIL</td>
<td>Remove from longword queue at tail, interlocked</td>
</tr>
<tr>
<td></td>
<td>The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.</td>
</tr>
<tr>
<td>REMQTILR</td>
<td>Remove from longword queue at tail, interlocked resident</td>
</tr>
<tr>
<td></td>
<td>The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation. This instruction requires that the queue be memory-resident and that the queue header and elements are quadword-aligned.</td>
</tr>
<tr>
<td>REMQTIQ</td>
<td>Remove from quadword queue at tail, interlocked</td>
</tr>
<tr>
<td></td>
<td>The self-relative queue entry preceding the header, pointed to by R16, is removed from the queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation and Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| REMQTIQR   | Remove from quadword queue at tail, interlocked resident  
The queue entry preceding the header, pointed to by R16, is removed from the self-relative queue and the address of the removed entry is returned in R1. The removal is interlocked to prevent concurrent interlocked insertions or removals at the head or tail of the same queue by another process, in a multiprocessor environment. The removal is a noninterruptible operation.  
This instruction requires that the queue be memory-resident and that the queue header and elements are octaword-aligned. |
| REMQUEL    | Remove from longword queue  
The queue entry addressed by R16 for REMQUEL or the entry addressed by the longword addressed by R16 for REMQUEL/D is removed from the longword absolute queue, and the address of the removed entry is returned in R1. The removal is a noninterruptible operation. |
| REMQUEQ    | Remove from quadword queue  
The queue entry addressed by R16 for REMQUEQ or the entry addressed by the quadword addressed by R16 for REMQUEL/D is removed from the quadword absolute queue, and the address of the removed entry removed is returned in R1. The removal is a noninterruptible operation. |
| RCCC       | Read system cycle counter  
Register R0 is written with the value of the system cycle counter. This counter is an unsigned 64-bit integer that increments at the same rate as the process cycle counter.  
The system cycle counter is suitable for timing a general range of intervals to within 10% error and may be used for detailed performance characterization. |
| SWASTEN    | Swap AST enable  
SWASTEN swaps the AST enable bit for the current mode. The new state for the enable bit is supplied in register R16<0> and previous state of the enable bit is returned, zero-extended, in R0.  
A check is made to determine if an AST is pending. If the enabling conditions are present for an AST at the completion of this instruction, the AST occurs before the next instruction. |
| WRITE_UNQ  | Write unique context  
WRITE UNQ writes the hardware process (thread) unique context value passed in R16 to internal storage or to the hardware privileged context block. |
| WR_PS_SW   | Write processor status software field  
WR_PS_SW writes the Processor Status software field (PS<SW>) with the low-order three bits of R16<2:0>. |
9.2 Privileged OpenVMS Alpha Palcode Instructions

The privileged PAL code instructions can be called in kernel mode only.

Table 9–2 describes the privileged OpenVMS Alpha PAL code instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFLUSH</td>
<td>Cache flush</td>
</tr>
<tr>
<td></td>
<td>At least the entire physical page specified by a page frame number in R16 is flushed from any data caches associated with the current processor. After doing a CFLUSH, the first subsequent load on the same processor to an arbitrary address in the target page is fetched from physical memory.</td>
</tr>
<tr>
<td>CSERVE</td>
<td>Console service</td>
</tr>
<tr>
<td></td>
<td>CSERVE is specific to each PAL code and console implementation and is not intended for operating system use.</td>
</tr>
<tr>
<td>DRAINA</td>
<td>Drain aborts</td>
</tr>
<tr>
<td></td>
<td>DRAINA stalls instruction issuing until all prior instructions are guaranteed to complete without incurring aborts.</td>
</tr>
<tr>
<td>HALT</td>
<td>Halt processor</td>
</tr>
<tr>
<td></td>
<td>The HALT instruction stops normal instruction processing.</td>
</tr>
<tr>
<td>LDQP</td>
<td>Load quadword physical</td>
</tr>
<tr>
<td></td>
<td>The quadword-aligned memory operand, whose physical address is in R16, is fetched and written to R0.</td>
</tr>
<tr>
<td></td>
<td>If the operand address in R16 is not quadword-aligned, the result is UNPREDICTABLE.</td>
</tr>
<tr>
<td>MFPR</td>
<td>Move from processor register</td>
</tr>
<tr>
<td></td>
<td>The internal processor register specified by the PAL code function field is written to R0.</td>
</tr>
<tr>
<td>MTPR</td>
<td>Move to processor register</td>
</tr>
<tr>
<td></td>
<td>The source operands in integer registers R16 (and R17, reserved for future use) are written to the internal processor register specified by the PAL code function field. The effect of loading a processor register is guaranteed to be active on the next instruction.</td>
</tr>
<tr>
<td>STQP</td>
<td>Store quadword physical</td>
</tr>
<tr>
<td></td>
<td>The quadword contents of R17 are written to the memory location whose physical address is in R16.</td>
</tr>
<tr>
<td></td>
<td>If the operand address in R16 is not quadword-aligned, the result is UNPREDICTABLE.</td>
</tr>
</tbody>
</table>
### Table 9–2 (Cont.): Privileged OpenVMS Alpha PALcode Instructions Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWPCTX</td>
<td>Swap privileged context</td>
</tr>
<tr>
<td></td>
<td>The SWPCTX instruction returns ownership of the data structure that contains the current hardware privileged context (the HWPCB) to the operating system and passes ownership of the new HWPCB to the processor.</td>
</tr>
<tr>
<td>SWPPAL</td>
<td>Swap PALcode image</td>
</tr>
<tr>
<td></td>
<td>SWPPAL causes the current PALcode to be replaced by the specified new PALcode image. Intended for use by operating systems only during bootstraps and by consoles during transitions to console I/O mode.</td>
</tr>
<tr>
<td>WTINT</td>
<td>Wait for interrupt</td>
</tr>
<tr>
<td></td>
<td>The WTINT instruction requests that, if possible, the PALcode wait for the first of either of the following conditions before returning: any interrupt other than a clock tick; or, the first clock tick after a specified number of clock ticks has been skipped.</td>
</tr>
</tbody>
</table>
The following sections specify the Privileged Architecture Library (PALcode) instructions that are required to support a Digital UNIX system.

10.1 Unprivileged Digital UNIX PALcode Instructions

Table 10–1 describes the unprivileged Digital UNIX PALcode instructions.

Table 10–1: Unprivileged Digital UNIX PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpt</td>
<td>Break point trap</td>
</tr>
<tr>
<td></td>
<td>The bpt instruction switches mode to kernel, builds a stack frame on the kernel stack, and dispatches to the breakpoint code.</td>
</tr>
<tr>
<td>bugchk</td>
<td>Bugcheck</td>
</tr>
<tr>
<td></td>
<td>The bugchk instruction switches mode to kernel, builds a stack frame on the kernel stack, and dispatches to the breakpoint code.</td>
</tr>
<tr>
<td>callsys</td>
<td>System call</td>
</tr>
<tr>
<td></td>
<td>The callsys instruction switches mode to kernel, builds a callsys stack frame, and dispatches to the system call code.</td>
</tr>
<tr>
<td>clrfen</td>
<td>Clear floating-point enable</td>
</tr>
<tr>
<td></td>
<td>The clrfen instruction writes a zero to the floating-point enable register.</td>
</tr>
<tr>
<td>gentrap</td>
<td>Generate trap</td>
</tr>
<tr>
<td></td>
<td>The gentrap instruction switches mode to kernel, builds a stack frame on the kernel stack, and dispatches to the gentrap code.</td>
</tr>
<tr>
<td>imb</td>
<td>I-stream memory barrier</td>
</tr>
<tr>
<td></td>
<td>The imb instruction makes the I-cache coherent with main memory.</td>
</tr>
<tr>
<td>rdunique</td>
<td>Read unique</td>
</tr>
<tr>
<td></td>
<td>The rdunique instruction returns the process unique value.</td>
</tr>
<tr>
<td>urti</td>
<td>Return from user mode trap</td>
</tr>
<tr>
<td></td>
<td>The urti instruction pops from the user stack the registers a0 through a2, the global pointer, the new user assembler temporary register, the stack pointer, the program counter, and the processor status register.</td>
</tr>
<tr>
<td>wrunique</td>
<td>Write unique</td>
</tr>
<tr>
<td></td>
<td>The wrunique instruction sets the process unique register.</td>
</tr>
</tbody>
</table>
10.2 Privileged Digital UNIX PALcode Instructions

The privileged PALcode instructions can be called only from kernel mode. They provide an interface to control the privileged state of the machine.

Table 10–2 describes the privileged Digital UNIX PALcode instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cflush</td>
<td>Cache flush</td>
</tr>
<tr>
<td></td>
<td>The cflush instruction flushes an entire physical page pointed to by the specified page frame number (PFN) from any data caches associated with the current processor. All processors must implement this instruction.</td>
</tr>
<tr>
<td>cserve</td>
<td>Console service</td>
</tr>
<tr>
<td></td>
<td>This instruction is specific to each PALcode and console implementation and is not intended for operating system use.</td>
</tr>
<tr>
<td>draina</td>
<td>Drain aborts</td>
</tr>
<tr>
<td></td>
<td>The draina instruction stalls instruction issuing until all prior instructions are guaranteed to complete without incurring aborts.</td>
</tr>
<tr>
<td>halt</td>
<td>Halt processor</td>
</tr>
<tr>
<td></td>
<td>The halt instruction stops normal instruction processing. Depending on the halt action setting, the processor can either enter console mode or the restart sequence.</td>
</tr>
<tr>
<td>rdmcies</td>
<td>Read machine check error summary</td>
</tr>
<tr>
<td></td>
<td>The rdmcies instruction returns the MCES register in v0.</td>
</tr>
<tr>
<td>rdps</td>
<td>Read processor status</td>
</tr>
<tr>
<td></td>
<td>The rdps instruction returns the current PS.</td>
</tr>
<tr>
<td>rdusp</td>
<td>Read user stack pointer</td>
</tr>
<tr>
<td></td>
<td>The rdusp instruction reads the user stack pointer while in kernel mode and returns it.</td>
</tr>
<tr>
<td>rdval</td>
<td>Read system value</td>
</tr>
<tr>
<td></td>
<td>The rdval instruction reads a 64-bit per-processor value and returns it.</td>
</tr>
<tr>
<td>retsys</td>
<td>Return from system call</td>
</tr>
<tr>
<td></td>
<td>The retsys instruction pops the return address, the user stack pointer, and the user global pointer from the kernel stack. It then saves the kernel stack pointer, sets mode to user, enables interrupts, and jumps to the address popped off the stack.</td>
</tr>
<tr>
<td>rti</td>
<td>Return from trap, fault or interrupt</td>
</tr>
<tr>
<td></td>
<td>The rti instruction pops certain registers from the kernel stack. If the new mode is user, the kernel stack is saved and the user stack restored.</td>
</tr>
</tbody>
</table>
Table 10–2 (Cont.): Privileged Digital UNIX PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>swpctx</td>
<td>Swap privileged context</td>
</tr>
<tr>
<td></td>
<td>The swpctx instruction saves the current process data in the current process control block (PCB). Then swpctx switches to the PCB and loads the new process context.</td>
</tr>
<tr>
<td>swpipl</td>
<td>Swap IPL</td>
</tr>
<tr>
<td></td>
<td>The swpipl instruction returns the current value IPL and sets the IPL.</td>
</tr>
<tr>
<td>swppal</td>
<td>Swap PAL code image</td>
</tr>
<tr>
<td></td>
<td>The swppal instruction causes the current PAL code to be replaced by the specified new PAL code image. Intended only for use by operating systems during bootstraps and by consoles during transitions to console I/O mode.</td>
</tr>
<tr>
<td>tbi</td>
<td>TB invalidate</td>
</tr>
<tr>
<td></td>
<td>The tbi instruction removes entries from the instruction and data translation buffers when the mapping entries change.</td>
</tr>
<tr>
<td>whami</td>
<td>Who_Am_I</td>
</tr>
<tr>
<td></td>
<td>The whami instruction returns the processor number for the current processor. The processor number is in the range 0 to the number of processors minus one (0..numproc–1) that can be configured in the system.</td>
</tr>
<tr>
<td>wrent</td>
<td>Write system entry address</td>
</tr>
<tr>
<td></td>
<td>The wrent instruction sets the virtual address of the system entry points.</td>
</tr>
<tr>
<td>wrfen</td>
<td>Write floating-point enable</td>
</tr>
<tr>
<td></td>
<td>The wrfen instruction writes a bit to the floating-point enable register.</td>
</tr>
<tr>
<td>wrripr</td>
<td>Write interprocessor interrupt request</td>
</tr>
<tr>
<td></td>
<td>The wrripr instruction generates an interprocessor interrupt on the processor number passed as an input parameter. The interrupt request is recorded on the target processor and initiated when the proper enabling conditions are present.</td>
</tr>
<tr>
<td>wrkgp</td>
<td>Write kernel global pointer</td>
</tr>
<tr>
<td></td>
<td>The wrkgp instruction writes the kernel global pointer internal register.</td>
</tr>
<tr>
<td>wrmcres</td>
<td>Write machine check error summary</td>
</tr>
<tr>
<td></td>
<td>The wrmcres instructions clears the machine check in progress bit and clears the processor- or system-correctable error in progress bit in the MCES register. The instruction also sets or clears the processor- or system-correctable error reporting enable bit in the MCES register.</td>
</tr>
<tr>
<td>wrperfmon</td>
<td>Performance monitoring function</td>
</tr>
<tr>
<td></td>
<td>The wrperfmon instruction alerts any performance monitoring software/hardware in the system to monitor the performance of this process.</td>
</tr>
</tbody>
</table>
Table 10–2 (Cont.): Privileged Digital UNIX PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrusp</td>
<td>Write user stack pointer</td>
</tr>
<tr>
<td></td>
<td>The wrusp instruction writes a value to the user stack pointer while in kernel mode.</td>
</tr>
<tr>
<td>wrval</td>
<td>Write system value</td>
</tr>
<tr>
<td></td>
<td>The wrval instruction writes a 64-bit per-processor value.</td>
</tr>
<tr>
<td>wrvptptr</td>
<td>Write virtual page table pointer</td>
</tr>
<tr>
<td></td>
<td>The wrvptptr instruction writes a pointer to the virtual page table pointer (vptptr).</td>
</tr>
<tr>
<td>wtint</td>
<td>Wait for interrupt</td>
</tr>
<tr>
<td></td>
<td>The wtint instruction requests that, if possible, the PALcode wait for the first of either of the following conditions before returning: any interrupt other than a clock tick; or, the first clock tick after a specified number of clock ticks has been skipped.</td>
</tr>
</tbody>
</table>
The following sections specify the Privileged Architecture Library (PALcode) instructions that are required to support a Windows NT Alpha system.

### 11.1 Unprivileged Windows NT Alpha PALcode Instructions

The unprivileged PALcode instructions provide support for system operations and may be called from both kernel and user modes.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpt</td>
<td>Breakpoint trap (standard user-mode breakpoint)</td>
</tr>
<tr>
<td></td>
<td>The bpt instruction raises a breakpoint general exception to the kernel, setting a USER_BREAKPOINT breakpoint type.</td>
</tr>
<tr>
<td>callkd</td>
<td>Call kernel debugger</td>
</tr>
<tr>
<td></td>
<td>The callkd instruction raises a breakpoint general exception to the kernel, setting the breakpoint type with the value supplied as an input parameter.</td>
</tr>
<tr>
<td>callsys</td>
<td>System service call</td>
</tr>
<tr>
<td></td>
<td>The callsys instruction raises a system service call exception to the kernel. Callsys switches to kernel mode if necessary, builds a trap frame on the kernel stack, and then enters the kernel at the kernel system service exception handler.</td>
</tr>
<tr>
<td>gentrap</td>
<td>Generate a trap</td>
</tr>
<tr>
<td></td>
<td>The gentrap instruction generates a software general exception that raises an exception code to the current thread. The exception code is generated from a trap number that is specified as an input parameter. Gentrap is used to raise software-detected exceptions such as bound check errors or overflow conditions.</td>
</tr>
<tr>
<td>imb</td>
<td>Instruction memory barrier</td>
</tr>
<tr>
<td></td>
<td>The imb instruction guarantees that all subsequent instruction stream fetches are coherent with respect to main memory. Imb must be issued before executing code in memory that has been modified (either by stores from the processor or DMA from an I/O processor). User-mode code that modifies the I-stream must call the appropriate Windows NT API to ensure I-cache coherency.</td>
</tr>
</tbody>
</table>
Table 11–1 (Cont.): Unprivileged Windows NT Alpha PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kbpt</td>
<td>Kernel breakpoint trap</td>
</tr>
<tr>
<td></td>
<td>The kbpt instruction raises a breakpoint general exception to the kernel, setting a KERNEL_BREAKPOINT breakpoint type.</td>
</tr>
<tr>
<td>rdteb</td>
<td>Read thread environment block pointer</td>
</tr>
<tr>
<td></td>
<td>The rdteb instruction returns the contents of the TEB internal processor register for the currently executing thread (the base address of the thread environment block).</td>
</tr>
</tbody>
</table>

11.2 Privileged Windows NT Alpha PALcode Instruction Summary

The privileged PALcode instructions provide support for system operations and may be called from only kernel mode.

Table 11–2: Privileged Windows NT Alpha PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csir</td>
<td>Clear software interrupt request</td>
</tr>
<tr>
<td></td>
<td>The csir instruction clears the specified bit in the SIRR internal processor register.</td>
</tr>
<tr>
<td>di</td>
<td>Disable all interrupts</td>
</tr>
<tr>
<td></td>
<td>The di instruction disables all interrupts by clearing the interrupt enable (IE) bit in the PSR internal processor register. The IRQL field is unaffected. Interrupts may be re-enabled via the ei instruction.</td>
</tr>
<tr>
<td>draina</td>
<td>Drain all aborts including machine checks</td>
</tr>
<tr>
<td></td>
<td>The draina instruction drains all aborts, including machine checks, from the current processor. Draina guarantees that no abort is signaled for an instruction issued before the draina while any instruction issued subsequent to the draina is executing.</td>
</tr>
<tr>
<td>dtbis</td>
<td>Data translation buffer invalidate single</td>
</tr>
<tr>
<td></td>
<td>The dtbis instruction invalidates a single data stream translation. The translation for the virtual address must be invalidated in all data translation buffers and in all virtual data caches.</td>
</tr>
<tr>
<td>ei</td>
<td>Enable interrupts</td>
</tr>
<tr>
<td></td>
<td>The ei instruction enables interrupts for the IRQL set in the PSR internal processor register by setting the interrupt enable (IE) bit in the PSR.</td>
</tr>
<tr>
<td>halt</td>
<td>Halt the operating system by forcing illegal instruction trap</td>
</tr>
<tr>
<td></td>
<td>The halt instruction forces an illegal instruction exception.</td>
</tr>
</tbody>
</table>
Table 11–2 (Cont.): Privileged Windows NT Alpha PALcode Instruction Summary

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initpal</td>
<td>Initialize PALcode data structures with operating system values. The initpal instruction is called early in the kernel initialization sequence to establish values for internal processor registers (IPRs) that are needed for trap and fault handling. The KGP and PCR registers are initialized once and persist throughout the run time of the operating system.</td>
</tr>
<tr>
<td>rdirql</td>
<td>Read the current IRQL from the PSR. The rdirql instruction returns the contents of the interrupt request level (IRQL) field of the PSR internal processor register.</td>
</tr>
<tr>
<td>rdksp</td>
<td>Read initial kernel stack pointer for the current thread. The rdksp instruction returns the contents of the IKSP (initial kernel stack pointer) internal processor register for the currently executing thread.</td>
</tr>
<tr>
<td>rdmces</td>
<td>Read the machine check error summary register. The rdmces instruction returns the contents of the machine check error summary (MCES) internal processor register.</td>
</tr>
<tr>
<td>rdpcr</td>
<td>Read the processor control region base address. The rdpcr instruction returns the contents of the PCR internal processor register (the base address value of the processor control region).</td>
</tr>
<tr>
<td>rdpsr</td>
<td>Read the current processor status register (PSR). The rdpsr instruction returns the contents of the current PSR (Processor Status Register) internal processor register.</td>
</tr>
<tr>
<td>rdthread</td>
<td>Read the thread value for the current thread. The rdthread instruction returns the contents of the THREAD internal processor register (the value of the currently executing thread).</td>
</tr>
<tr>
<td>reboot</td>
<td>Transfer to console firmware. The reboot instruction stops the operating system from executing and returns execution to the boot environment. Reboot is responsible for completing the ARC restart block before returning to the boot environment.</td>
</tr>
<tr>
<td>restart</td>
<td>Restart the operating system from the restart block. The restart instruction restores saved processor state and resumes execution of the operating system.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation and description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------</td>
</tr>
</tbody>
</table>
| retsys   | Return from system service call exception  
The retsys instruction returns from a system service call exception by unwinding the trap frame and returning to the code stream that was executing when the original exception was initiated. In addition, retsys accepts a parameter to set software interrupt requests that became pending while the exception was handled. |
| rfe      | Return from trap or interrupt  
The rfe instruction returns from exceptions by unwinding the trap frame and returning to the code stream that was executing when the original exception was initiated. In addition, rfe accepts a parameter to set software interrupt requests that became pending while the exception was handled. |
| ssir     | Set software interrupt request  
The ssir instruction sets software interrupt requests by setting the appropriate bits in the SIRR internal processor register. |
| swpctx   | Swap thread context  
The swpctx instruction swaps the privileged portions of thread context. Thread context is swapped by establishing the new IKSP, THREAD, and TEB internal processor register values. |
| swpirql  | Swap the current interrupt request level  
The swpirql instruction swaps the current IRQL field in the PSR internal processor register by setting the processor so that only permitted interrupts are enabled for the new IRQL. Swpirql updates the IRQL field and returns the previous IRQL. |
| swpksp   | Swap the initial kernel stack pointer for the current thread  
The swpksp instruction returns the value of the previous IKSP internal processor register and writes a new IKSP for the currently executing thread. |
| swppal   | Swap the currently executing PAL code  
The swppal instruction swaps the currently executing PAL code by transferring to the base address of the new PAL code image in the PAL code environment. |
| swpprocess | Swap process context (swap address space)  
The swpprocess instruction swaps the privileged process context by changing the address space for the currently executing thread. |
| tbia     | Translation buffer invalidate all  
The tbia instruction invalidates all translations and virtual cache blocks within the processor. |
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation and description</th>
</tr>
</thead>
</table>
| tbis     | Translation buffer invalidate single  
The tbis instruction invalidates a single virtual translation. The translation for the passed virtual address must be invalidated in all processor translation buffers and virtual caches. |
| tbisasn  | Translation buffer invalidate single for ASN  
The tbisasn instruction invalidates a single virtual translation for a specified address space. The translation for the passed virtual address must be invalidated in all processor translation buffers and virtual caches. |
| wrentry  | Write kernel exception entry routine  
The wrentry instruction provides the registry of exception handling routines for the exception classes. The kernel must use wrentry to register an exception handler for each of the exception classes. |
| wrmces   | Write the machine check error summary register  
The wrmces instruction writes new values for the MCES internal processor register and returns the previous values of that register. |
| wrperfmon| Write performance counter interrupt control information  
The wrperfmon instruction writes control information for the two processor performance counters. One parameter identifies the selected performance counter, while another controls whether the selected performance counter is enabled or disabled. The instruction returns the previous enable state for the selected performance counter. |
A.1 Hardware-Software Compact

The Alpha architecture, like all RISC architectures, depends on careful attention to data alignment and instruction scheduling to achieve high performance.

Since there will be various implementations of the Alpha architecture, it is not obvious how compilers can generate high-performance code for all implementations. This chapter gives some scheduling guidelines that, if followed by all compilers and respected by all implementations, will result in good performance. As such, this section represents a good-faith compact between hardware designers and software writers. It represents a set of common goals, not a set of architectural requirements. Thus, an Appendix, not a Chapter.

Many of the performance optimizations discussed below are advantageous only for frequently executed code. For rarely executed code, they may produce a bigger program that is not any faster. Some of the branching optimizations also depend on good prediction of which path from a conditional branch is more frequently executed. These optimizations are best done by using an execution profile, either an estimate generated by compiler heuristics, or a real profile of a previous run, such as that gathered by PC-sampling in PCA.

Each computer architecture has a “natural word size.” For the PDP-11, it is 16 bits; for VAX, 32 bits; and for Alpha, 64 bits. Other architectures also have a natural word size that varies between 16 and 64 bits. Except for very low-end implementations, ALU data paths, cache access paths, chip pin buses, and main memory data paths are all usually the natural word size.

As an architecture becomes commercially successful, high-end implementations inevitably move to double-width data paths that can transfer an aligned (at an even natural word address) pair of natural words in one cycle. For Alpha, this means 128-bit wide data paths will eventually be implemented. It is difficult to get much speed advantage from paired transfers unless the code being executed has instructions and data appropriately aligned on aligned octaword boundaries. Since this is difficult to retrofit to old code, the following sections sometimes encourage “over-aligning” to octaword boundaries in anticipation of high-speed Alpha implementations.

In some cases, there are performance advantages to aligning instructions or data to cache-block boundaries, or putting data whose use is correlated into the same cache block, or trying to avoid cache conflicts by not having data whose use is correlated placed at addresses that are equal modulo the cache size. Since the Alpha architecture will have many implementations, an exact cache design cannot be outlined here. Nonetheless, some expected bounds can be stated.
1. Small (first-level) cache sizes will likely be in the range 2 KB to 64 KB
2. Small cache block sizes will likely be 16, 32, 64, or 128 bytes
3. Large (second- or third-level) cache sizes will likely be in the range 128 KB to 8 MB
4. Large cache block sizes will likely be 32, 64, 128, or 256 bytes
5. TB sizes will likely be in the range 16 to 1024 entries

Thus, if two data items need to go in different cache blocks, it is desirable to make them at least 128 bytes apart (modulo 2 KB). Doing so creates a high probability of allowing both items to be in a small cache simultaneously for all Alpha implementations.

In each case below, the performance implication is given by an order-of-magnitude number: 1, 3, 10, 30, or 100. A factor of 10 means that the performance difference being discussed will likely range from 3 to 30 across all Alpha implementations.

A.2 Instruction-Stream Considerations

The following sections describe considerations for the instruction stream.

A.2.1 Instruction Alignment

Code PSECTs should be octaword aligned. Targets of frequently taken branches should be at least quadword aligned, and octaword aligned for very frequent loops. Compilers could use execution profiles to identify frequently taken branches.

Most Alpha implementations will fetch aligned quadwords of instruction stream (two instructions), and many will waste an instruction-issue cycle on a branch to an odd longword. High-end implementations may eventually fetch aligned octawords, and waste up to three issue cycles on a branch to an odd longword. Some implementations may only be able to fetch wide chunks of instructions every other CPU cycle. Fetching four instructions from an aligned octaword can get at most one cache miss, while fetching them from an odd longword address can get two or even three cache misses.

Quadword I-fetch implementors should give first priority to executing aligned quadwords quickly. Octaword-fetch implementors should give first priority to executing aligned octawords quickly, and second priority to executing aligned quadwords quickly. Dual-issue implementations should give first priority to issuing both halves of an aligned quadword in one cycle, and second priority to buffering and issuing other combinations.

A.2.2 Multiple Instruction Issue — Factor of 3

Some Alpha implementations will issue multiple instructions in a single cycle. To improve the odds of multiple-issue, compilers should choose pairs of instructions to put in aligned quadwords. Pick one from column A and one from column B (but only a total of one load/store/branch per pair).
<table>
<thead>
<tr>
<th>Column A</th>
<th>Column B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Operate</td>
<td>Floating Operate</td>
</tr>
<tr>
<td>Floating Load/Store</td>
<td>Integer Load/Store</td>
</tr>
<tr>
<td>Floating Branch</td>
<td>Integer Branch</td>
</tr>
<tr>
<td></td>
<td>BR/BSR/J SR</td>
</tr>
</tbody>
</table>

Implementors of multiple-issue machines should give first priority to dual-issuing at least the above pairs, and second priority to multiple-issue of other combinations.

In general, the above rules will give a good hardware-software match, but compilers may want to implement model-specific switches to generate code tuned more exactly to a specific implementation.

A.2.3 Branch Prediction and Minimizing Branch-Taken — Factor of 3

In many Alpha implementations, an unexpected change in I-stream address will result in about 10 lost instruction times. “Unexpected” may mean any branch-taken or may mean a mispredicted branch. In many implementations, even a correctly predicted branch to a quadword target address will be slower than straight-line code.

Compilers should follow these rules to minimize unexpected branches:

1. Implementations will predict all forward conditional branches as not-taken, and all backward conditional branches as taken. Based on execution profiles, compilers should physically rearrange code so that it has matching behavior.

2. Make basic blocks as big as possible. A good goal is 20 instructions on average between branch-taken. This means unrolling loops so that they contain at least 20 instructions, and putting subroutines of less than 20 instructions directly in line. It also means using execution profiles to rearrange code so that the frequent case of a conditional branch falls through. For very high-performance loops, it will be profitable to move instructions across conditional branches to fill otherwise wasted instruction issue slots, even if the instructions moved will not always do useful work. Note that the Conditional Move instructions can sometimes be used to avoid breaking up basic blocks.

3. In an if-then-else construct whose execution profile is skewed even slightly away from 50%-50% (51-49 is enough), put the infrequent case completely out of line, so that the frequent case encounters zero branch-takens, and the infrequent case encounters two branch-takens. If the infrequent case is rare (5%), put it far enough away that it never comes into the I-cache. If the infrequent case is extremely rare (error message code), put it on a page of rarely executed code and expect that page never to be paged in.

4. There are two functionally identical branch-format opcodes, BSR and BR.
Compilers should use the first one for subroutine calls, and the second for GOTOs. Some implementations may push a stack of predicted return addresses for BSR and not push the stack for BR. Failure to compile the correct opcode will result in mispredicted return addresses, and hence make subroutine returns slow.

5. The memory-format JSR instruction has 16 unused bits. These should be used by the compilers to communicate a hint about expected branch-target behavior (see Common Architecture, Chapter 4).

If the JSR is used for a computed GOTO or a CASE statement, compile bits <15:14> as 00, and bits <13:0> such that (updated PC+Instr<13:0>*4) <15:0> equals (likely_target_addr) <15:0>. In other words, pick the low 14 bits so that a normal PC+displacement*4 calculation will match the low 16 bits of the most likely target longword address. (Implementations will likely prefetch from the matching cache block.)

If the JSR is used for a computed subroutine call, compile bits <15:14> as 01, and bits <13:0> as above. Some implementations will prefetch the call target using the prediction and also push updated PC on a return-prediction stack.

If the JSR is used as a subroutine return, compile bits <15:14> as 10. Some implementations will pop an address off a return-prediction stack.

If the JSR is used as a coroutine linkage, compile bits <15:14> as 11. Some implementations will pop an address off a return-prediction stack and also push updated PC on the return-prediction stack.

Implementors should give first priority to executing straight-line code with no branch-takens as quickly as possible, second priority to predicting conditional branches based on the sign of the displacement field (backward taken, forward not-taken), and third priority to predicting subroutine return addresses by running a small prediction stack. (VAX traces show a stack of two to four entries correctly predicts most branches.)
A.2.4 Improving I-Stream Density — Factor of 3

Compilers should try to use profiles to make sure almost 100% of the bytes brought into an I-cache are actually executed. This means aligning branch targets and putting rarely executed code out of line. Doing so would consistently make an I-cache appear about two times larger, compared to current VAX practice.

The example below shows the bytes actually brought into a VAX cache (from part of an address trace of a DLINPAC). The dots represent bytes brought into the cache but never executed. They occupy about half of the cache.

Each line shows the use of an aligned 64-byte I-cache block. A portion of DLINPAC and a portion of OpenVMS Alpha 4.x are shown. Uppercase I is the first byte of an instruction, and lowercase i marks subsequent bytes. Period (.) shows a byte brought into the cache but never executed.

<table>
<thead>
<tr>
<th>I-fetch</th>
<th>Byte 0</th>
<th>Byte 63</th>
</tr>
</thead>
</table>
| 000268C0 | .........................IiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiIiI

A.2.5 Instruction Scheduling — Factor of 3

The performance of Alpha programs is sensitive to how carefully the code is scheduled to minimize instruction-issue delays.

"Result latency" is defined as the number of CPU cycles that must elapse between an instruction that writes a result register and one that uses that register, if execution-time stalls are to be avoided. Thus, a latency of zero means that the instruction writes a result register and the instruction that uses that register can be multiple-issued in the same cycle. A latency of 2 means that if the writing instruction is issued at cycle N, the reading instruction can issue no earlier than cycle N+2. Latency is implementation specific.

Most Alpha instructions have a non-zero result latency. Compilers should schedule code so that a result is not used too soon, at least in frequently executed code (inner
loops, as identified by execution profiles). In general, this will require loop unrolling and short procedure inlining.

Assume that implementations can dual-issue instructions. Assume that Load and JSR instructions have a latency of 3, shifts and byte manipulation a latency of 2, integer multiply a latency of 10, and other integer operates a latency of 1. Assume floating multiply has a latency of 5, floating divide a latency of 10, and other floating operates a latency of 4. Scheduling to these latencies gives at least reasonable performance on current implementations.

Compilers should try to schedule code to match the above latency rules and also to match the multiple-issue rules. If doing both is impractical for a particular sequence of code, the latency rules are more important (since they apply even in single-issue implementations).

Implementors should give first priority to minimizing the latency of back-to-back integer operations, of address calculations immediately followed by load/store, of load immediately followed by branch, and of compare immediately followed by branch. Second priority should be given to minimizing latencies in general.

A.3 Data-Stream Considerations

The following sections describe considerations for the data stream.

A.3.1 Data Alignment — Factor of 10

Data PSECTs should be at least octaword aligned, so that aggregates (arrays, some records, subroutine stack frames) can be allocated on aligned octaword boundaries to take advantage of any implementations with aligned octaword data paths, and to decrease the number of cache fills in almost all implementations.

Aggregates (arrays, records, common blocks, and so forth) should be allocated on at least aligned octaword boundaries whenever language rules allow. In some implementations, a series of writes that completely fill a cache block may be a factor of 10 faster than a series of writes that partially fill a cache block, when that cache block would give a read miss. This is true of write-back caches that read a partially filled cache block from memory, but optimize away the read for completely filled blocks.

For such implementations, long strings of sequential writes will be faster if they start on a cache-block boundary (a multiple of 128 bytes will do well for most, if not all, Alpha implementations). This applies to array results that sweep through large portions of memory, and also to register-save areas for context switching, graphics frame buffer accesses, and other places where exactly 8, 16, 32, or more quadwords are stored sequentially. Allocating the targets at multiples of 8, 16, 32, or more quadwords, respectively, and doing the writes in order of increasing address will maximize the write speed.

Items within aggregates that are forced to be unaligned (records, common blocks) should generate compile-time warning messages and inline byte extract/insert code. Users must be educated that the warning message means that they are taking a factor of 30 performance hit.
Compilers should consider supplying a switch that allows the compiler to pad aggregates to avoid unaligned data.

Compiled code for parameters should assume that the parameters are aligned. Unaligned actuals will therefore cause run-time alignment traps and very slow fixups. The fixup routine, if invoked, should generate warning messages to the user, preferably giving the first few statement numbers that are doing unaligned parameter access, and at the end of a run the total number of alignment traps (and perhaps an estimate of the performance improvement if the data were aligned). Again, users must be educated that the trap routine warning message means they are taking a factor of 30 performance hit.

Frequently used scalars should reside in registers. Each scalar datum allocated in memory should normally be allocated an aligned quadword to itself, even if the datum is only a byte wide. This allows aligned quadword loads and stores and avoids partial-quadword writes (which may be half as fast as full-quadword writes, due to such factors as read-modify-write a quadword to do quadword ECC calculation).

Implementors should give first priority to fast reads of aligned octawords and second priority to fast writes of full cache blocks. Partial-quadword writes need not have a fast repetition rate.

### A.3.2 Shared Data in Multiple Processors — Factor of 3

Software locks are aligned quadwords and should be allocated to large cache blocks that either contain no other data, or read-mostly data whose usage is correlated with the lock.

Whenever there is high contention for a lock, one processor will have the lock and be using the guarded data, while other processors will be in a read-only spin loop on the lock bit. Under these circumstances, any write to the cache block containing the lock will likely cause excess bus traffic and cache fills, thus having a performance impact on all processors that are involved, and the buses between them. In some decomposed FORTRAN programs, refills of the cache blocks containing one or two frequently used locks can account for a third of all the bus bandwidth the program consumes.

Whenever there is almost no contention for a lock, one processor will have the lock and be using the guarded data. Under these circumstances, it might be desirable to keep the guarded data in the same cache block as the lock.

For the high-sharing case, compilers should assume that almost all accesses to shared data result in cache misses all the way back to main memory, for each distinct cache block used. Such accesses will likely be a factor of 30 slower than cache hits. It is helpful to pack correlated shared data into a small number of cache blocks. It is helpful also to segregate blocks written by one processor from blocks read by others.

Therefore, accesses to shared data, including locks, should be minimized. For example, a four-processor decomposition of some manipulation of a 1000-row array should avoid accessing lock variables every row, but instead might access a lock variable every 250 rows.
Array manipulation should be partitioned across processors so that cache blocks do not thrash between processors. Having each of four processors work on every fourth array element severely impairs performance on any implementation with a cache block of four elements or larger. The processors all contend for copies of the same cache blocks and use only one quarter of the data in each block. Writes in one processor severely impair cache performance on all processors.

A better decomposition is to give each processor the largest possible contiguous chunk of data to work on (N/4 consecutive rows for four processors and row-major array storage; N/4 columns for column-major storage). With the possible exception of three cache blocks at the partition boundaries, this decomposition will result in each processor caching data that is touched by no other processor.

Operating-system scheduling algorithms should attempt to minimize process migration from one processor to another. Any time migration occurs, there are likely to be a large number of cache misses on the new processor.

Similarly, operating-system scheduling algorithms should attempt to enforce some affinity between a given device's interrupts and the processor on which the interrupt-handler runs. I/O control data structures and locks for different devices should be disjoint. Doing both of these allows higher cache hit rates on the corresponding I/O control data structures.

Implementors should give first priority to an efficient (low-bandwidth) way of transferring isolated lock values and other isolated, shared write data between processors.

Implementors should assume that the amount of shared data will continue to increase, so over time the need for efficient sharing implementations will also increase.

A.3.3 Avoiding Cache/TB Conflicts — Factor of 1

Occasionally, programs that run with a direct-mapped cache or TB will thrash, taking excessive cache or TB misses. With some work, thrashing can be minimized at compile time.

In a frequently executed loop, compilers could allocate the data items accessed from memory so that, on each loop iteration, all of the memory addresses accessed are either in exactly the same aligned 64-byte block, or differ in bits VA<10:6>. For loops that go through arrays in a common direction with a common stride, this means allocating the arrays, checking that the first-iteration addresses differ, and if not, inserting up to 64 bytes of padding between the arrays. This rule will avoid thrashing in small direct-mapped data caches with block sizes up to 64 bytes and total sizes of 2K bytes or more.

Example:

```fortran
REAL*4 A(1000), B(1000)
DO 60 i=1, 1000
  60 A(i) = f(B(i))
```

A–8  Alpha Architecture Handbook
BAD allocation (A and B thrash in 8 KB direct-mapped cache):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4K</th>
<th>8K</th>
<th>12K</th>
<th>16K</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BETTER allocation (A and B offset by 64 mod 2 KB, so 16 elements of A and 16 of B can be in cache simultaneously):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4K</th>
<th>8K+64</th>
<th>12K</th>
<th>16K</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BEST allocation (A and B offset by 64 mod 2 KB, so 16 elements of A and 16 of B can be in cache simultaneously, and both arrays fit entirely in 8 KB or bigger cache):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4K-64</th>
<th>8K</th>
<th>12K</th>
<th>16K</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a frequently executed loop, compilers could allocate the data items accessed from memory so that, on each loop iteration, all of the memory addresses accessed are either in exactly the same 8 KB page, or differ in bits VA<17:13>. For loops that go through arrays in a common direction with a common stride, this means allocating the arrays, checking that the first-iteration addresses differ, and if not, inserting up to 8K bytes of padding between the arrays. This rule will avoid thrashing in direct-mapped TBs and in some large direct-mapped data caches, with total sizes of 32 pages (256 KB) or more.

Usually, this padding will mean zero extra bytes in the executable image, just a skip in virtual address space to the next-higher page boundary.

For large caches, the rule above should be applied to the I-stream, in addition to all the D-stream references. Some implementations will have combined I-stream /D-stream large caches.

Both of the rules above can be satisfied simultaneously, thus often eliminating thrashing in all anticipated direct-mapped cache/TB implementations.

**A.3.4 Sequential Read/Write — Factor of 1**

All other things being equal, sequences of consecutive reads or writes should use ascending (rather than descending) memory addresses. Where possible, the memory address for a block of 2**K bytes should be on a 2**K boundary, since this minimizes
the number of different cache blocks used and minimizes the number of partially written cache blocks.

To avoid overrunning memory bandwidth, sequences of more than eight quadword load or store instructions should be broken up with intervening instructions (if there is any useful work to be done).

For consecutive reads, implementors should give first priority to prefetching ascending cache blocks, and second priority to absorbing up to eight consecutive quadword load instructions (aligned on a 64-byte boundary) without stalling.

For consecutive writes, implementors should give first priority to avoiding read overhead for fully written aligned cache blocks, and second priority to absorbing up to eight consecutive quadword store instructions (aligned on a 64-byte boundary) without stalling.

A.3.5 Prefetching — Factor of 3

To use FETCH and FETCH_M effectively, software should follow this programming model:

1. Assume that at most two FETCH instructions can be outstanding at once, and that there are two prefetch address registers, PREa and PREb, to hold prefetching state. FETCH instructions alternate between loading PREa and PREb. Each FETCH instruction overwrites any previous prefetching state, thus terminating any previous prefetch that is still in progress in the register that is loaded. The order of fetching within a block and the order between PREa and PREb are UNPREDICTABLE.

   Implementation Note:
   Implementations are encouraged to alternate at convenient intervals between PREa and PREb.

2. Assume, for maximum efficiency, that there should be about 64 unrelated memory access instructions (load or store) between a FETCH and the first actual data access to the prefetched data.

3. Assume, for instruction-scheduling purposes in a multilevel cache hierarchy, that FETCH does not prefetch data to the innermost cache level, but rather one level out. Schedule loads to bury the last level of misses.

4. Assume that FETCH is worthwhile if, on average, at least half the data in a block will be accessed. Assume that FETCH_M is worthwhile if, on average, at least half the data in a block will be modified.

5. Treat FETCH as a vector load. If a piece of code could usefully prefetch four operands, launch the first two prefetches, do about 128 memory references worth of work, then launch the next two prefetches, do about 128 more memory references worth of work, then start using the four sets of prefetched data.

6. Treat FETCH as having the same effect on a cache as a series of 64 quadword loads. If the loads would displace useful data, so will FETCH. If two sets of loads
from specific addresses will thrash in a direct-mapped cache, so will two FETCH
instructions using the same pair of addresses.

Implementation Note:
Hardware implementations are expected to provide either no support for
FETCHx or support that closely matches this model.

A.4 Code Sequences

The following section describes code sequences.

A.4.1 Aligned Byte/Word Memory Accesses

The instruction sequences given in Common Architecture, Chapter 4, for byte and
word accesses are worst-case code. Further, they do not reflect the instructions
available with the BWX extension, described in the Preface and Appendix D. If the
BWX extension instructions are available, it is wise to consider them rather than the
sequences that follow; the following sequences are appropriate if the BWX extension
instructions are not available.

In the common case of accessing a byte or aligned word field at a known offset from
a pointer that is expected to be at least longword aligned, the common-case code is
much shorter.

“Expected” means that the code should run fast for a longword-aligned pointer and
trap for unaligned. The trap handler may at its option fix up the unaligned reference.

For access at a known offset D from a longword-aligned pointer Rx, let D.lw be D
rounded down to a multiple of 4 ((D div 4)*4), and let D.mod be D mod 4.

In the common case, the intended sequence for loading and zero-extending an aligned
word is:

```
LDL R1,D.lw(Rx) ! Traps if unaligned
EXTWL R1,#D.mod,R1 ! Picks up word at byte 0 or byte 2
```

In the common case, the intended sequence for loading and sign-extending an aligned
word is:

```
LDL R1,D.lw(Rx) ! Traps if unaligned
SLL R1,#48-8*D.mod,R1 ! Aligns word at high end of R1
SRA R1,#48,R1 ! SEXT to low end of R1
```

Note:
The shifts often can be combined with shifts that might surround subsequent
arithmetic operations (for example, to produce word overflow from the high end
of a register).

In the common case, the intended sequence for loading and zero-extending a byte is:

```
LDL R1,D.lw(Rx) !
EXTBL R1,#D.mod,R1 !
```
In the common case, the intended sequence for loading and sign-extending a byte is:

\[
\begin{align*}
\text{LDL} & \quad \text{R1,D.1w(Rx)} \\
\text{SLL} & \quad \text{R1,#56-8\times D.mod,R1} \\
\text{SRA} & \quad \text{R1,#56,R1}
\end{align*}
\]

In the common case, the intended sequence for storing an aligned word R5 is:

\[
\begin{align*}
\text{LDL} & \quad \text{R1,D.1w(Rx)} \\
\text{INSWL} & \quad \text{R5,#D.mod,R3} \\
\text{MSKWL} & \quad \text{R1,#D.mod,R1} \\
\text{BIS} & \quad \text{R3,R1,R1} \\
\text{STL} & \quad \text{R1,D.1w(Rx)}
\end{align*}
\]

In the common case, the intended sequence for storing a byte R5 is:

\[
\begin{align*}
\text{LDL} & \quad \text{R1,D.1w(Rx)} \\
\text{INSBL} & \quad \text{R5,#D.mod,R3} \\
\text{MSKBL} & \quad \text{R1,#D.mod,R1} \\
\text{BIS} & \quad \text{R3,R1,R1} \\
\text{STL} & \quad \text{R1,D.1w(Rx)}
\end{align*}
\]

### A.4.2 Division

In all implementations, floating-point division is likely to have a substantially longer result latency than floating-point multiply; in addition, in many implementations multiplies will be pipelined and divides will not.

Thus, any division by a constant power of two should be compiled as a multiply by the exact reciprocal, if it is representable without overflow or underflow. If language rules or surrounding context allow, other divisions by constants can be closely approximated via multiplication by the reciprocal.

Integer division does not exist as a hardware opcode. Division by a constant can always be done via UMULH of another appropriate constant, followed by a right shift. General quadword division by true variables can be done via a subroutine. The subroutine could test for small divisors (less than about 1000 in absolute value) and for those, do a table lookup on the exact constant and shift count for an UMULH /shift sequence. For the remaining cases, a table lookup on about a 1000-entry table and a multiply can give a linear approximation to 1/divisor that is accurate to 16 bits.

Using this approximation, a multiply and a back-multiply and a subtract can generate one 16-bit quotient “digit” plus a 48-bit new partial dividend. Three more such steps can generate the full quotient. Having prior knowledge of the possible sizes of the divisor and dividend, normalizing away leading bytes of zeros, and performing an early-out test can reduce the average number of multiplies to about five (compared to a best case of one and a worst case of nine).

### A.4.3 Byte Swap

When it is necessary to swap all the bytes of a datum, perhaps because the datum originated on a machine of the opposite byte numbering convention, the simplest sequence is to use the VAX floating-point load instruction to swap words, followed by an integer sequence to swap four pairs of bytes. Assume as shown below that an
aligned quadword datum is in memory at location X and is to be left in R1 after byte-swapping; temp is an aligned quadword temporary, and “.” (period) in the comments stands for a byte of zeros. Similar sequences can be used for data in registers, sometimes doing the byte swaps first and word swap second:

; X = ABCD EFGH
LDG F0,X ; F0 = GHEF CDAB
STT F0,temp
LDQ R1,temp ; R1 = GHEF CDAB
SLL R1,#8,R2 ; R2 = HEFC DAB.
SRL R1,#8,R1 ; R1 = .GHE FCDA
ZAP R2,#55(hex),R2 ; R2 = H.F. D.B.
ZAP R1,#AA(hex),R1 ; R1 = .G.E .C.A
OR R1,R2,R1 ; R1 = HGFE DCBA

For bulk swapping of arrays, this sequence can be usefully unrolled about four times and scheduled, using four different aligned quadword memory temps.

A.4.4 Stylized Code Forms

Using the same stylized code form for a common operation makes compiler output a little more readable and makes it more likely that an implementation will speed up the stylized form.

A.4.4.1 NOP

The universal NOP form is:

\[
\text{UNOP} \equiv \text{LDQ} \_U \ R31,0(Rx)
\]

In most implementations, UNOP should encounter no operand issue delays, no destination issue delay, and no functional unit issue delays. (In some implementations, it may encounter an operand issue delay for Rx.) Implementations are free to optimize UNOP into no action and zero execution cycles.

If the actual instruction is encoded as LDQ \_U Rn,0(Rx), where n is other than 31, and such an instruction generates a memory-management exception, it is UNPREDICTABLE whether UNOP would generate the same exception. On most implementations, UNOP does not generate memory management exceptions.

The standard NOP forms are:

\[
\begin{align*}
\text{NOP} & \equiv \text{BIS} \ R31,R31,R31 \\
\text{FNOP} & \equiv \text{CPYS} \ F31,F31,F31
\end{align*}
\]

These generate no exceptions. In most implementations, they should encounter no operand issue delays and no destination issue delay. Implementations are free to optimize these into no action and zero execution cycles.

A.4.4.2 Clear a Register

The standard clear register forms are:

\[
\begin{align*}
\text{CLR} & \equiv \text{BIS} \ R31,R31,Rx \\
\text{FCLR} & \equiv \text{CPYS} \ F31,F31,Fx
\end{align*}
\]
These generate no exceptions. In most implementations, they should encounter no operand issue delays, and no functional unit issue delay.

A.4.4.3 Load Literal

The standard load integer literal (ZEXT 8-bit) form is:

\[
\text{MOV} \ #\text{lit8}, \text{Ry} \Rightarrow \text{BIS} \ R31, \text{lit8}, \text{Ry}
\]

The Alpha literal construct in Operate instructions creates a canonical longword constant for values 0..255.

A longword constant stored in an Alpha 64-bit register is in canonical form when bits \(<63:32>=bit <31>\).

A canonical 32-bit literal can usually be generated with one or two instructions, but sometimes three instructions are needed. Use the following procedure to determine the offset fields of the instructions:

\[
\text{val} = <\text{sign-extended, 32-bit value}>
\]

\[
\text{low} = \text{val}<15:0>
\]

\[
\text{tmp1} = \text{val} - \text{SEXT}(\text{low}) \quad ! \text{Account for LDA instruction}
\]

\[
\text{high} = \text{tmp1}<31:16>
\]

\[
\text{tmp2} = \text{tmp1} - \text{SHIFT}\_LEFT(\text{SEXT}(\text{high}, 16) )
\]

\[
\text{if tmp2 NE 0 then}
\]

\[
! \text{original val was in range }7FFFF8000_{16}.7FFFFFFF_{16}
\]

\[
\text{extra} = 4000_{16}
\]

\[
\text{tmp1} = \text{tmp1} - 40000000_{16}
\]

\[
\text{high} = \text{tmp1}<31:16>
\]

\[
\text{else}
\]

\[
\text{extra} = 0
\]

\[
\text{endif}
\]

The general sequence is:

\[
\text{LDA } \text{Rdst}, \text{low(R31)}
\]

\[
\text{LDAH } \text{Rdst}, \text{extra(Rdst)} \quad ! \text{Omit if extra=0}
\]

\[
\text{LDAH } \text{Rdst}, \text{high(Rdst)} \quad ! \text{Omit if high=0}
\]

A.4.4.4 Register-to-Register Move

The standard register move forms are:

\[
\text{MOV} \ RX,RY \Rightarrow \text{BIS} \ RX,RX,RY
\]

\[
\text{FMOV} \ FX,FY \Rightarrow \text{CPYS} \ FX,FX,FY
\]

These generate no exceptions. In most implementations, these should encounter no functional unit issue delay.

A.4.4.5 Negate

The standard register negate forms are:

\[
\text{NEGz} \ Rx,Ry \Rightarrow \text{SUBz} \ R31,Rx,Ry \quad ! z = L \text{ or } Q
\]

\[
\text{NEGz} \ Fx,Fy \Rightarrow \text{SUBz} \ F31,Fx,Fy \quad ! z = F G S \text{ or } T
\]

\[
\text{FNEGz} \ Fx,Fy \Rightarrow \text{CPYSN} \ Fx,Fx,Fy \quad ! z = F G S \text{ or } T
\]
The integer subtract generates no Integer Overflow trap if Rx contains the largest negative number (SUBz/V would trap). The floating subtract generates a floating-point exception for a non-finite value in Fx. The CPYSN form generates no exceptions.

A.4.4.6 NOT
The standard integer register NOT form is:

\[ \text{NOT } Rx, Ry == \text{ ORNOT } R31, Rx, Ry \]

This generates no exceptions. In most implementations, this should encounter no functional unit issue delay.

A.4.4.7 Booleans
The standard alternative to BIS is:

\[ \text{OR } Rx, Ry, Rz == \text{ BIS } Rx, Ry, Rz \]

The standard alternative to BIC is:

\[ \text{ANDNOT } Rx, Ry, Rz == \text{ BIC } Rx, Ry, Rz \]

The standard alternative to EQV is:

\[ \text{XORNOT } Rx, Ry, Rz == \text{ EQV } Rx, Ry, Rz \]

A.4.5 Trap Barrier
The TRAPB instruction guarantees that it and any following instructions do not issue until all possible preceding traps have been signaled. This does not mean that all preceding instructions have necessarily run to completion (for example, a Load instruction may have passed all the fault checks but not yet delivered data from a cache miss).

A.4.6 Pseudo-Operations (Stylized Code Forms)
This section summarizes the pseudo-operations for the Alpha architecture that may be used by various software components in an Alpha system. Most of these forms are discussed in preceding sections.

In the context of this section, pseudo-operations all represent a single underlying machine instruction. Each pseudo-operation represents a particular instruction with either replicated fields (such as FMOV), or hard-coded zero fields. Since the pattern is distinct, these pseudo-operations can be decoded by instruction decode mechanisms.

In Table A–1, the pseudo-operation codes can be viewed as macros with parameters. The formal form is listed in the left column, and the expansion in the code stream listed in the right column.

Some instruction mnemonics have synonyms. These are different from pseudo-operations in that each synonym represents the same underlying instruction with no special encoding of operand fields. As a result, synonyms cannot be distinguished
from each other. They are not listed in the table that follows. Examples of synonyms are: BIC/ANDNOT, BIS/OR, and EQV/XORNTO.

Table A–1: Decodable Pseudo-Operations (Stylized Code Forms)

<table>
<thead>
<tr>
<th>Pseudo-Operation in Listing</th>
<th>Actual Instruction Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-exception generic floating absolute value:</td>
<td></td>
</tr>
<tr>
<td>FABS Fx, Fy CPYS F31, Fx, Fy</td>
<td></td>
</tr>
<tr>
<td>Branch to target (21-bit signed displacement):</td>
<td></td>
</tr>
<tr>
<td>BR target BR R31, target</td>
<td></td>
</tr>
<tr>
<td>Clear integer register:</td>
<td></td>
</tr>
<tr>
<td>CLR Rx BIS R31, R31, Rx</td>
<td></td>
</tr>
<tr>
<td>Clear a floating-point register:</td>
<td></td>
</tr>
<tr>
<td>FCLR Fx CPYS F31, F31, Fx</td>
<td></td>
</tr>
<tr>
<td>Floating-point move:</td>
<td></td>
</tr>
<tr>
<td>FMOV Fx, Fy CPYS Fx, Fx, Fy</td>
<td></td>
</tr>
<tr>
<td>No-exception generic floating negation:</td>
<td></td>
</tr>
<tr>
<td>FNEG Fx, Fy CPYSN Fx, Fx, Fy</td>
<td></td>
</tr>
<tr>
<td>Floating-point no-op:</td>
<td></td>
</tr>
<tr>
<td>FNOP CPYS F31, F31, F31</td>
<td></td>
</tr>
<tr>
<td>Move Rx/8-bit zero-extended literal to Ry:</td>
<td></td>
</tr>
<tr>
<td>MOV {Rx/Lit8}, Ry BIS R31, {Rx/Lit8}, Ry</td>
<td></td>
</tr>
<tr>
<td>Move 16-bit sign-extended literal to Rx:</td>
<td></td>
</tr>
<tr>
<td>MOV Lit, Rx LDA Rx, lit(R31)</td>
<td></td>
</tr>
<tr>
<td>Move to FPCR:</td>
<td></td>
</tr>
<tr>
<td>MT_FPCR Fx MT_FPCR Fx, Fx, Fx</td>
<td></td>
</tr>
<tr>
<td>Move from FPCR:</td>
<td></td>
</tr>
<tr>
<td>MF_FPCR Fx MF_FPCR Fx, Fx, Fx</td>
<td></td>
</tr>
<tr>
<td>Negate F_floating:</td>
<td></td>
</tr>
<tr>
<td>NEGF Fx, Fy SUBF F31, Fx, Fy</td>
<td></td>
</tr>
<tr>
<td>Negate F_floating, semi-precise:</td>
<td></td>
</tr>
<tr>
<td>NEGF/S Fx, Fy SUBF/S F31, Fx, Fy</td>
<td></td>
</tr>
<tr>
<td>Negate G_floating:</td>
<td></td>
</tr>
<tr>
<td>NEGG Fx, Fy SUBG F31, Fx, Fy</td>
<td></td>
</tr>
<tr>
<td>Negate G_floating, semi-precise:</td>
<td></td>
</tr>
</tbody>
</table>

A–16 Alpha Architecture Handbook
### Table A–1 (Cont.): Decodable Pseudo-Operations (Stylized Code Forms)

<table>
<thead>
<tr>
<th>Pseudo-Operation in Listing</th>
<th>Actual Instruction Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEGG/S Fx, Fy</td>
<td>SUBG/S F31, Fx, Fy</td>
</tr>
<tr>
<td>Negate longword:</td>
<td></td>
</tr>
<tr>
<td>NEGL {Rx/Lit8}, Ry</td>
<td>SUBL R31, {Rx/Lit}, Ry</td>
</tr>
<tr>
<td>Negate longword with overflow detection:</td>
<td></td>
</tr>
<tr>
<td>NEGL/V {Rx/Lit8}, Ry</td>
<td>SUBL/V R31, {Rx/Lit}, Ry</td>
</tr>
<tr>
<td>Negate quadword:</td>
<td></td>
</tr>
<tr>
<td>NEGQ {Rx/Lit8}, Ry</td>
<td>SUBQ R31, {Rx/Lit}, Ry</td>
</tr>
<tr>
<td>Negate quadword with overflow detection:</td>
<td></td>
</tr>
<tr>
<td>NEGQ/V {Rx/Lit8}, Ry</td>
<td>SUBQ/V R31, {Rx/Lit}, Ry</td>
</tr>
<tr>
<td>Negate S_floating:</td>
<td></td>
</tr>
<tr>
<td>NEGS Fx, Fy</td>
<td>SUBS F31, Fx, Fy</td>
</tr>
<tr>
<td>Negate S_floating, software with underflow detection:</td>
<td></td>
</tr>
<tr>
<td>NEGS/SU Fx, Fy</td>
<td>SUBS/SU F31, Fx, Fy</td>
</tr>
<tr>
<td>Negate S_floating, software with underflow and inexact result detection:</td>
<td></td>
</tr>
<tr>
<td>NEGS/SUI Fx, Fy</td>
<td>SUBS/SUI F31, Fx, Fy</td>
</tr>
<tr>
<td>Negate T_floating:</td>
<td></td>
</tr>
<tr>
<td>NEGT Fx, Fy</td>
<td>SUBT F31, Fx, Fy</td>
</tr>
<tr>
<td>Negate T_floating, software with underflow detection:</td>
<td></td>
</tr>
<tr>
<td>NEGT/SU Fx, Fy</td>
<td>SUBT/SU F31, Fx, Fy</td>
</tr>
<tr>
<td>Negate T_floating, software with underflow and inexact result detection:</td>
<td></td>
</tr>
<tr>
<td>NEGT/SUI Fx, Fy</td>
<td>SUBT/SUI F31, Fx, Fy</td>
</tr>
<tr>
<td>Integer no-op:</td>
<td></td>
</tr>
<tr>
<td>NOP BIS R31, R31, R31</td>
<td></td>
</tr>
<tr>
<td>Logical NOT of Rx/8-bit zero-extended literal storing results in Ry:</td>
<td></td>
</tr>
<tr>
<td>NOT {Rx/Lit8}, Ry</td>
<td>ORNOT R31, {Rx/Lit}, Ry</td>
</tr>
<tr>
<td>Longword sign-extension of Rx storing results in Ry:</td>
<td></td>
</tr>
<tr>
<td>SEXTL {Rx/Lit8}, Ry</td>
<td>ADDL R31, {Rx/Lit}, Ry</td>
</tr>
<tr>
<td>Universal NOP for both integer and floating-point code:</td>
<td></td>
</tr>
<tr>
<td>UNOP LDQ_U R31,0(Rx)</td>
<td></td>
</tr>
</tbody>
</table>

**Software Considerations**  A–17
A.5 Timing Considerations: Atomic Sequences

A sufficiently long instruction sequence between LDx_L and STx_C will never complete, because periodic timer interrupts will always occur before the sequence completes. The following rules describe sequences that will eventually complete in all Alpha implementations:

1. At most 40 operate or conditional-branch (not taken) instructions executed in the sequence between LDx_L and STx_C.

2. At most two I-stream TB-miss faults. Sequential instruction execution guarantees this.

3. No other exceptions triggered during the last execution of the sequence.

Implementation Note:

On all expected implementations, this allows for about 50 μsec of execution time, even with 100 percent cache misses. This should satisfy any requirement for a 1 msec timer interrupt rate.
Appendix B
IEEE Floating-Point Conformance

A subset of IEEE Standard for Binary Floating-Point Arithmetic (754-1985) is provided in the Alpha floating-point instructions. This appendix describes how to construct a complete IEEE implementation.

The order of presentation parallels the order of the IEEE specification.

B.1 Alpha Choices for IEEE Options

Alpha supports IEEE single, double, and optionally (in software) extended double formats. There is no hardware support for the optional extended double format.

Alpha hardware supports normal and chopped IEEE rounding modes. IEEE plus infinity and minus infinity rounding modes can be implemented in hardware or software.

Alpha hardware does not support optional IEEE software trap enable/disable modes; see the following discussion about software support.

Alpha hardware supports add, subtract, multiply, divide, convert between floating formats, convert between floating and integer formats, and compare. Software routines support square root, remainder, round to integer in floating-point format, and convert binary to/from decimal.

In the Alpha architecture, copying without change of format is not considered an operation. (LDx, CPYSx, and STx do not check for non-finite numbers; an operation would.) Compilers may generate ADDx F31,Fx,Fy to get the opposite effect.

Optional operations for differing formats are not provided.

The Alpha choice is that the accuracy provided by conversions between decimal strings and binary floating-point numbers will meet or exceed IEEE standard requirements. It is implementation dependent whether the software binary/decimal conversions beyond 9 or 17 digits treat any excess digits as zeros.

Overflow and underflow, NaNs, and infinities encountered during software binary to decimal conversion return strings that specify the conditions. Such strings can be truncated to their shortest unambiguous length.

Alpha hardware supports comparisons of same-format numbers. Software supports comparisons of different-format numbers.

In the Alpha architecture, results are true-false in response to a predicate.

Alpha hardware supports the required six predicates and the optional unordered predicate. The other 19 optional predicates can be constructed from sequences of two comparisons and two branches.
Except for the compare instructions (CMPTyy) and the Overflow Disable (OVFD) option, Alpha hardware supports infinity arithmetic by trapping. That is the case when an infinity operand is encountered and when an infinity is to be created from finite operands by overflow or division by zero. A software trap handler (interposed between the hardware and the IEEE user) provides correct infinity arithmetic.

Except for the Invalid Operation Disable (INVD) option, Alpha hardware supports NaNs by trapping when a NaN operand is encountered and when a NaN is to be created. A software trap handler (interposed between the hardware and the IEEE user) provides correct Signaling and Quiet NaN behavior.

In the Alpha architecture, Quiet NaNs do not afford retrospective diagnostic information.

In the Alpha architecture, copying a Signaling NaN without a change of format does not signal an invalid exception (LDx, CPYSx, and STx do not check for non-finite numbers). Compilers may generate ADDx.F31,Fx,Fy to get the opposite effect.

Alpha hardware fully supports negative zero operands, and follows the IEEE rules for creating negative zero results.

Except for the optional trap disable bits in the FPCR, Alpha hardware does not supply IEEE exception trap behavior; the hardware traps are a superset of the IEEE-required conditions. A software trap handler (interposed between the hardware and the IEEE user) provides correct IEEE exception behavior.

In the Alpha architecture, tininess is detected by hardware after rounding, and loss of accuracy is detected by software as an inexact result.

In the Alpha architecture, user trap handlers are supported by compilers and a software trap handler (interposed between the hardware and the IEEE user), as described in the next section.

B.2 Alpha Hardware Support of Software Exception Handlers

Except for the optional trap disable bits in the FPCR, the hardware trap behavior of Alpha instructions is determined at compile time; short of recompiling, there are no dynamic facilities for changing hardware trap behavior.

B.2.1 Choosing Degrees of IEEE Compliance

There is an essential disparity between the Alpha design goal of fast execution and the IEEE design goal of exact trap behavior. The Alpha hardware architecture provides means for users to choose various degrees of IEEE compliance, at appropriate performance cost.

Instructions compiled without the /Software modifier cannot produce IEEE-compliant trap or status bit behavior, nor can they provide IEEE-compliant non-finite arithmetic. Trapping and stopping on non-finite operands or results (rather than the IEEE default of continuing with NaNs propagated) is an Alpha value-added behavior that some users prefer.
Instructions compiled without the /Underflow hardware trap enable modifier cannot produce IEEE-compliant underflow trap or status bit behavior, nor can they provide IEEE-compliant denormal results. They are fast and provide true zero (not minus zero) results whenever underflow occurs. This is an Alpha value-added behavior that some users prefer.

Instructions compiled without the /Inexact hardware trap enable modifier cannot produce IEEE-compliant inexact trap or status bit behavior. Except when the Inexact Disable (INED) option is implemented, trapping on inexact is painfully slow. Few users appear to prefer enabling inexact trapping, but they can get it if they really want it.

Except when the optional Overflow Disable (OVFD), Division by Zero Disable (DZED), or Invalid Operation Disable (INVD) bits in the FPCR are set, IEEE floating-point instructions compiled with the /Software enable modifier produce hardware traps and unpredictable values for overflow, division by zero, or invalid operation. A software trap handler may then produce the chosen IEEE-required behavior. The software trap handler reports an enabled IEEE exception to the user application as a fault, rather than as a trap. Because the exception is reported as a fault, the reported PC points to the trigger instruction, rather than to a point after the trigger instruction.

Regardless of whether or not an enabled fault occurs, the software completion handler sets the result register and the status flags to the IEEE standard nontrapping result, as further defined in the IEEE Standard section in Common Architecture, Chapter 4.

Except when the optional Underflow Disable (UNFD) bit in the FPCR is set, IEEE floating-point instructions compiled with the /Software enable and /Underflow enable modifiers produce hardware traps and true zero values for underflow; a software trap handler may then produce all IEEE-required behavior. Such instructions with /Software and /Underflow enabled, but without an underflow condition that produce zero value results, always have the correct sign.

IEEE floating-point instructions compiled with the /Inexact enable modifier produce hardware traps that allow a software trap handler to produce all IEEE-required behavior.

Thus, to get full IEEE compliance of all the required features of the standard, users must compile with all three options enabled.

To get the optional full IEEE user trap handler behavior, a software trap handler must be provided that implements the five exception flags, dynamic user trap handler disabling, handler saving and restoring, default behavior for disabled user trap handlers, and linkages that allow a user handler to return a substitute result.

The software trap handler uses the FP_Control quadword, along with the floating-point control register (FPCR), to provide various levels of IEEE-compliant behavior.
B.2.2 IEEE Floating-Point Control (FP_C) Quadword

Operating system implementations provide the following support for an IEEE floating-point control quadword (FP_C), illustrated in Figure B–1 and described in Table B–1.

Figure B–1: IEEE Floating-Point Control (FP_C) Quadword

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63–48</td>
<td>Reserved for implementation software.</td>
</tr>
<tr>
<td>47–22</td>
<td>Reserved for future architecture definition.</td>
</tr>
<tr>
<td>21</td>
<td>Inexact Result Status (INES)</td>
</tr>
<tr>
<td></td>
<td>A floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.</td>
</tr>
</tbody>
</table>
## Table B–1 (Cont.): Floating-Point Control (FP_C) Quadword Bit Summary

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Underflow Status (UNFS)</td>
</tr>
<tr>
<td></td>
<td>A floating arithmetic or conversion operation underflowed the destination exponent.</td>
</tr>
<tr>
<td>19</td>
<td>Overflow Status (OVFS)</td>
</tr>
<tr>
<td></td>
<td>A floating arithmetic or conversion operation overflowed the destination exponent.</td>
</tr>
<tr>
<td>18</td>
<td>Division by Zero Status (DZES)</td>
</tr>
<tr>
<td></td>
<td>An attempt was made to perform a floating divide operation with a divisor of zero.</td>
</tr>
<tr>
<td>17</td>
<td>Invalid Operation Status (INVS)</td>
</tr>
<tr>
<td></td>
<td>An attempt was made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values were illegal.</td>
</tr>
<tr>
<td>16–12</td>
<td>Reserved for implementation software.</td>
</tr>
<tr>
<td>11–6</td>
<td>Reserved for future architecture definition.</td>
</tr>
<tr>
<td>5</td>
<td>Inexact Result Enable (INEE)</td>
</tr>
<tr>
<td></td>
<td>Initiate an INE exception if a floating arithmetic or conversion operation gives a result that differs from the mathematically exact result.</td>
</tr>
<tr>
<td>4</td>
<td>Underflow Enable (UNFE)</td>
</tr>
<tr>
<td></td>
<td>Initiate a UNF exception if a floating arithmetic or conversion operation underflows the destination exponent.</td>
</tr>
<tr>
<td>3</td>
<td>Overflow Enable (OVFE)</td>
</tr>
<tr>
<td></td>
<td>Initiate an OVF exception if a floating arithmetic or conversion operation overflows the destination exponent.</td>
</tr>
<tr>
<td>2</td>
<td>Division by Zero Enable (DZEE)</td>
</tr>
<tr>
<td></td>
<td>Initiate a DZE exception if an attempt is made to perform a floating divide operation with a divisor of zero.</td>
</tr>
<tr>
<td>1</td>
<td>Invalid Operation Enable (INVE)</td>
</tr>
<tr>
<td></td>
<td>Initiate an INV exception if an attempt is made to perform a floating arithmetic, conversion, or comparison operation, and one or more of the operand values is illegal.</td>
</tr>
<tr>
<td>0</td>
<td>Reserved for implementation software.</td>
</tr>
</tbody>
</table>

## B.3 Mapping to IEEE Standard

There are five IEEE exceptions, each of which can be “IEEE software trap-enabled” or disabled (the default condition). Implementing the IEEE software trap-enabled mode is optional in the IEEE standard.

Our assumption, therefore, is that the only access to IEEE-specified software trap-enabled results will be generated in assembly language code. The following design allows this, but only if such assembly language code has TRAPB instructions after
each floating-point instruction, and generates the IEEE-specified scaled result in a trap handler by emulating the instruction that was trapped by hardware overflow/underflow detection, using the original operands.

There is a set of detailed IEEE-specified result values, both for operations that are specified to raise IEEE traps and those that do not. This behavior is created on Alpha by four layers of hardware, PAL code, the operating-system trap handler, and the user IEEE trap handler, as shown in Figure B–2.

Figure B–2: IEEE Trap Handling Behavior

The IEEE-specified trap behavior occurs only with respect to the user IEEE trap handler (the last layer in Figure B–2); any trap-and-fixup behavior in the first three layers is outside the scope of the IEEE standard.

The IEEE number system is divided into finite and non-finite numbers:

- The finites are normal numbers:
  - -MAX..-MIN, -0, 0, +MIN..+MAX

- The non-finites are:
  - Denormals, +/- Infinity, Signaling NaN, Quiet NaN

Alpha hardware must treat minus zero operands and results as special cases, as required by the IEEE standard.

Table B–2 specifies, for the IEEE /Software modes, which layer does each piece of trap handling. See Common Architecture, Chapter 4, for more detail on the hardware instruction descriptions.
<table>
<thead>
<tr>
<th>Alpha Instructions</th>
<th>Hardware</th>
<th>PAL</th>
<th>OS Trap Handler</th>
<th>User Software Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>FBEQ FBNE FBLT FBLE FBGT FBGE</td>
<td>Bits Only—No Exceptions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDS LDT</td>
<td>Bits Only—No Exceptions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STS STT</td>
<td>Bits Only—No Exceptions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPYS CPYSN</td>
<td>Bits Only—No Exceptions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMOVx</td>
<td>Bits Only—No Exceptions</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ADDx SUBx INPUT Exceptions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Hardware</th>
<th>PAL</th>
<th>OS Trap Handler</th>
<th>User Software Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denormal operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply sum</td>
<td>-</td>
</tr>
<tr>
<td>+/-Inf operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply sum</td>
<td>-</td>
</tr>
<tr>
<td>QNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>-</td>
</tr>
<tr>
<td>SNan operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td>+/-Inf + -Inf</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>[Invalid Op]</td>
</tr>
</tbody>
</table>

### ADDx SUBx OUTPUT Exceptions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Hardware</th>
<th>PAL</th>
<th>OS Trap Handler</th>
<th>User Software Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent overflow</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply +/-Inf</td>
<td>[Overflow] Scale by bias adjust</td>
</tr>
<tr>
<td>Exponent underflow and disabled</td>
<td>Supply +0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Exponent underflow and enabled</td>
<td>Supply +0 and trap</td>
<td>Trap</td>
<td>Supply +/-MIN denorm +/-0</td>
<td>[Underflow] Scale by bias adjust</td>
</tr>
<tr>
<td>Inexact and disabled</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inexact and enabled</td>
<td>Supply sum and trap</td>
<td>Trap</td>
<td>-</td>
<td>[Inexact]</td>
</tr>
</tbody>
</table>

1An implementation could choose instead to trap to PALcode and have the PALcode supply a zero result on all underflows.
<table>
<thead>
<tr>
<th>Alpha Instructions</th>
<th>Hardware</th>
<th>PAL</th>
<th>OS Trap Handler</th>
<th>User Software Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MULx INPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Denormal operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply prod.</td>
<td></td>
</tr>
<tr>
<td>+/-Inf operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply prod.</td>
<td></td>
</tr>
<tr>
<td>QNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td></td>
</tr>
<tr>
<td>SNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td>0 * Inf</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td><strong>MULx OUTPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exponent overflow</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply +/-Inf</td>
<td>[Overflow] Scale by bias adjust</td>
</tr>
<tr>
<td>Exponent underflow and disabled</td>
<td>Supply +0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Exponent underflow and enabled</td>
<td>Supply +0 and Trap</td>
<td>Trap</td>
<td>Supply +/-MIN denorm +/-0</td>
<td>[Underflow] Scale by bias adjust</td>
</tr>
<tr>
<td>Inexact and disabled</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inexact and enabled</td>
<td>Supply prod. and trap</td>
<td>Trap</td>
<td>-</td>
<td>[Inexact]</td>
</tr>
<tr>
<td><strong>DIVx INPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Denormal operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply quot.</td>
<td></td>
</tr>
<tr>
<td>+/-Inf operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply quot.</td>
<td></td>
</tr>
<tr>
<td>QNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td></td>
</tr>
<tr>
<td>SNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>[Invalid Op]</td>
</tr>
</tbody>
</table>
### Table B–2 (Cont.): IEEE Floating-Point Trap Handling

#### DIVx INPUT Exceptions

<table>
<thead>
<tr>
<th>Alpha Instructions</th>
<th>Hardware</th>
<th>PAL</th>
<th>OS Trap Handler</th>
<th>User Software Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVx INPUT Exceptions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0/0 or Inf/Inf</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td>A/0</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply +/-Inf</td>
<td>[Div. Zero]</td>
</tr>
</tbody>
</table>

#### DIVx OUTPUT Exceptions

| Exponent overflow | Trap | Trap | Supply +/-Inf   | [Overflow] Scale by bias adjust |
| Exponent underflow and disabled | Supply +0 | - | - | - |
| Exponent underflow and enabled | Supply +0 and trap | Trap | Supply +/-MIN denorm +/-0 | [Underflow] Scale by bias adjust |
| Inexact and disabled | - | - | - | - |
| Inexact and enabled | Supply quot. and trap | Trap | - | [Inexact] |

#### CMPTEQ CMPTUN INPUT Exceptions

| Denormal operand | Trap | Trap | Supply (≠) | - |
| QNaN operand     | Trap | Trap | Supply False for EQ, True for UN | - |
| SNaN operand     | Trap | Trap | Supply False/True | [Invalid Op] |

#### CMPTLT CMPTLE INPUT Exceptions

| Denormal operand | Trap | Trap | Supply ≤ or < | - |
| QNaN operand     | Trap | Trap | Supply False | [Invalid Op] |
Table B–2 (Cont.): IEEE Floating-Point Trap Handling

<table>
<thead>
<tr>
<th>Alpha Instructions</th>
<th>Hardware</th>
<th>PAL</th>
<th>OS Trap Handler</th>
<th>User Software Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMPTLT CMPTLE INPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply False</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td><strong>CVTfi INPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Denormal operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply Cvt</td>
<td>–</td>
</tr>
<tr>
<td>+/-Inf operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply 0</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td>QNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply 0</td>
<td>–</td>
</tr>
<tr>
<td>SNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply 0</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td><strong>CVTfi OUTPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inexact and disabled</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Inexact and enabled</td>
<td>Supply Cvt and trap</td>
<td>Trap</td>
<td>–</td>
<td>[Inexact]</td>
</tr>
<tr>
<td>Integer overflow</td>
<td>Supply Trunc. result and trap if enabled</td>
<td>Trap</td>
<td>–</td>
<td>[Invalid Op]</td>
</tr>
<tr>
<td><strong>CVTif OUTPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inexact and disabled</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Inexact and enabled</td>
<td>Supply Cvt and trap</td>
<td>Trap</td>
<td>–</td>
<td>[Inexact]</td>
</tr>
<tr>
<td><strong>CVTff INPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Denormal operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply Cvt</td>
<td>–</td>
</tr>
<tr>
<td>+/-Inf operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply Cvt</td>
<td>–</td>
</tr>
</tbody>
</table>

2An implementation could choose instead to trap to PALcode on extreme values and have the PALcode supply a truncated result on all overflows.
Table B–2 (Cont.): IEEE Floating-Point Trap Handling

<table>
<thead>
<tr>
<th>Alpha Instructions</th>
<th>Hardware</th>
<th>PAL</th>
<th>OS Trap Handler</th>
<th>User Software Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CVTff INPUT Exceptions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>-</td>
</tr>
<tr>
<td>SNaN operand</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply QNaN</td>
<td>[Invalid Op]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CVTff OUTPUT Exceptions</strong></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponent overflow</td>
<td>Trap</td>
<td>Trap</td>
<td>Supply +/-Inf</td>
<td>[Overflow] Scale by bias adjust</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+/-Inf</td>
<td></td>
</tr>
<tr>
<td>Exponent underflow and disabled</td>
<td>Supply +0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Exponent underflow and enabled</td>
<td>Supply +0 and trap</td>
<td>Trap</td>
<td>Supply +/-MIN denorm +/-0</td>
<td>[Underflow] Scale by bias adjust</td>
</tr>
<tr>
<td>Inexact and disabled</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inexact and enabled</td>
<td>Supply Cvt and trap</td>
<td>Trap</td>
<td>-</td>
<td>[Inexact]</td>
</tr>
</tbody>
</table>

Other IEEE operations (software subroutines or sequences of instructions), are listed here for completeness:

- Remainder
- SQRT
- Round float to integer-valued float
- Convert binary to/from decimal
- Compare, other combinations than the four above
Table B–3 shows the IEEE standard charts.

**Table B–3: IEEE Standard Charts**

<table>
<thead>
<tr>
<th>Exception</th>
<th>IEEE Software TRAP Disabled (IEEE Default)</th>
<th>IEEE Software TRAP Enabled (Optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invalid Operation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1) Input signaling NaN</td>
<td>Quiet NaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>(2) Mag. subtract Inf.</td>
<td>Quiet NaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>(3) 0 * Inf.</td>
<td>Quiet NaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>(4) 0/0 or Inf/Inf</td>
<td>Quiet NaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>(5) x REM 0 or Inf REM y</td>
<td>Quiet NaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>(6) SQRT(negative non-zero)</td>
<td>Quiet NaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>(7) Cvt to int(ovfl)</td>
<td>Low-order bits</td>
<td></td>
</tr>
<tr>
<td>(8) Cvt to int(Inv, NaN)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(9) Compare unordered</td>
<td>Quiet NaN</td>
<td></td>
</tr>
<tr>
<td><strong>Division by Zero</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x/0, x finite &lt;&gt;0</td>
<td>+/-Inf</td>
<td></td>
</tr>
<tr>
<td><strong>Overflow</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Round nearest</td>
<td>+/-Inf.</td>
<td>Res/2**192 or 1536</td>
</tr>
<tr>
<td>Round to zero</td>
<td>+/-MAX</td>
<td>Res/2**192 or 1536</td>
</tr>
<tr>
<td>Round to -Inf</td>
<td>+/-MAX/-Inf</td>
<td>Res/2**192 or 1536</td>
</tr>
<tr>
<td>Round to +Inf</td>
<td>+/-Inf/-MAX</td>
<td>Res/2**192 or 1536</td>
</tr>
<tr>
<td><strong>Underflow</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Underflow</td>
<td>0/denorm</td>
<td>Res*2**192 or 1536</td>
</tr>
<tr>
<td><strong>Inexact</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inexact</td>
<td>Rounded</td>
<td>Res</td>
</tr>
</tbody>
</table>
Appendix C
Instruction Summary

This appendix contains a summary of all instructions and opcodes in the Alpha architecture. All values are in hexadecimal radix.

C.1 Common Architecture Instruction Summary

This section contains a summary of all common Alpha instructions. Table C–1 describes the contents of the Format and Opcode columns in Table C–2.

Table C–1: Instruction Format and Opcode Notation

<table>
<thead>
<tr>
<th>Instruction Format</th>
<th>Format Symbol</th>
<th>Opcode Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>Bra</td>
<td>oo</td>
<td>oo is the 6-bit opcode field</td>
</tr>
<tr>
<td>Floating-point</td>
<td>F-P</td>
<td>oo.fff</td>
<td>oo is the 6-bit opcode field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fff is the 11-bit function code field</td>
</tr>
<tr>
<td>Memory</td>
<td>Mem</td>
<td>oo</td>
<td>oo is the 6-bit opcode field</td>
</tr>
<tr>
<td>Memory/func code</td>
<td>Mfc</td>
<td>oo.ffff</td>
<td>oo is the 6-bit opcode field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ffff is the 16-bit function code in the displacement field</td>
</tr>
<tr>
<td>Memory/branch</td>
<td>Mbr</td>
<td>oo.h</td>
<td>oo is the 6-bit opcode field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>h is the high-order two bits of the displacement field</td>
</tr>
<tr>
<td>Operate</td>
<td>Opr</td>
<td>oo.ff</td>
<td>oo is the 6-bit opcode field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ff is the 7-bit function code field</td>
</tr>
<tr>
<td>PAL code</td>
<td>Pcd</td>
<td>oo</td>
<td>oo is the 6-bit opcode field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>the particular PAL code instruction is specified in the 26-bit function code field</td>
</tr>
</tbody>
</table>

Qualifiers for operate format instructions are shown in Table C–2. Qualifiers for IEEE and VAX floating-point instructions are shown in Sections C–3 and C–4, respectively.

Table C–2: Common Architecture Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDDF</td>
<td>F-P</td>
<td>15.080</td>
<td>Add F_floating</td>
</tr>
<tr>
<td>ADDDG</td>
<td>F-P</td>
<td>15.0A0</td>
<td>Add G_floating</td>
</tr>
<tr>
<td>ADDDL</td>
<td>Opr</td>
<td>10.00</td>
<td>Add longword</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Format</td>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>--------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ADDL/V</td>
<td>Opr</td>
<td>10.40</td>
<td>Add quadword</td>
</tr>
<tr>
<td>ADDQ</td>
<td>Opr</td>
<td>10.20</td>
<td>Add quadword</td>
</tr>
<tr>
<td>ADDQ/V</td>
<td></td>
<td>10.60</td>
<td></td>
</tr>
<tr>
<td>ADDS</td>
<td>F-P</td>
<td>16.080</td>
<td>Add S_floating</td>
</tr>
<tr>
<td>ADDT</td>
<td>F-P</td>
<td>16.0A0</td>
<td>Add T_floating</td>
</tr>
<tr>
<td>AMASK</td>
<td>Opr</td>
<td>11.61</td>
<td>Architecture mask</td>
</tr>
<tr>
<td>AND</td>
<td>Opr</td>
<td>11.00</td>
<td>Logical product</td>
</tr>
<tr>
<td>BEQ</td>
<td>Bra</td>
<td>39</td>
<td>Branch if = zero</td>
</tr>
<tr>
<td>BGE</td>
<td>Bra</td>
<td>3E</td>
<td>Branch if ≥ zero</td>
</tr>
<tr>
<td>BGT</td>
<td>Bra</td>
<td>3F</td>
<td>Branch if &gt; zero</td>
</tr>
<tr>
<td>BIC</td>
<td>Opr</td>
<td>11.0</td>
<td>Bit clear</td>
</tr>
<tr>
<td>BIS</td>
<td>Opr</td>
<td>11.20</td>
<td>Logical sum</td>
</tr>
<tr>
<td>BLBC</td>
<td>Bra</td>
<td>38</td>
<td>Branch if low bit clear</td>
</tr>
<tr>
<td>BLBS</td>
<td>Bra</td>
<td>3C</td>
<td>Branch if low bit set</td>
</tr>
<tr>
<td>BLE</td>
<td>Bra</td>
<td>3B</td>
<td>Branch if ≤ zero</td>
</tr>
<tr>
<td>BLT</td>
<td>Bra</td>
<td>3A</td>
<td>Branch if &lt; zero</td>
</tr>
<tr>
<td>BNE</td>
<td>Bra</td>
<td>3D</td>
<td>Branch if ≠ zero</td>
</tr>
<tr>
<td>BR</td>
<td>Bra</td>
<td>30</td>
<td>Unconditional branch</td>
</tr>
<tr>
<td>BSR</td>
<td>Mbr</td>
<td>34</td>
<td>Branch to subroutine</td>
</tr>
<tr>
<td>CALL_PAL</td>
<td>Pcd</td>
<td>00</td>
<td>Trap to PAL code</td>
</tr>
<tr>
<td>CMOVEQ</td>
<td>Opr</td>
<td>11.24</td>
<td>CMOVE if = zero</td>
</tr>
<tr>
<td>CMOVGE</td>
<td>Opr</td>
<td>11.46</td>
<td>CMOVE if ≥ zero</td>
</tr>
<tr>
<td>CMOVGT</td>
<td>Opr</td>
<td>11.66</td>
<td>CMOVE if &gt; zero</td>
</tr>
<tr>
<td>CMOVLBC</td>
<td>Opr</td>
<td>11.16</td>
<td>CMOVE if low bit clear</td>
</tr>
<tr>
<td>CMOVLS</td>
<td>Opr</td>
<td>11.14</td>
<td>CMOVE if low bit set</td>
</tr>
<tr>
<td>CMOVLE</td>
<td>Opr</td>
<td>11.64</td>
<td>CMOVE if ≤ zero</td>
</tr>
<tr>
<td>CMOVLT</td>
<td>Opr</td>
<td>11.44</td>
<td>CMOVE if &lt; zero</td>
</tr>
<tr>
<td>CMOVNE</td>
<td>Opr</td>
<td>11.26</td>
<td>CMOVE if ≠ zero</td>
</tr>
<tr>
<td>CMPBGE</td>
<td>Opr</td>
<td>10.0F</td>
<td>Compare byte</td>
</tr>
<tr>
<td>CMPEQ</td>
<td>Opr</td>
<td>10.2D</td>
<td>Compare signed quadword equal</td>
</tr>
<tr>
<td>CMPGEQ</td>
<td>F-P</td>
<td>15.0A5</td>
<td>Compare G_floating equal</td>
</tr>
<tr>
<td>CMPGLE</td>
<td>F-P</td>
<td>15.0A7</td>
<td>Compare G_floating less than or equal</td>
</tr>
<tr>
<td>CMPGLT</td>
<td>F-P</td>
<td>15.0A6</td>
<td>Compare G_floating less than</td>
</tr>
<tr>
<td>CMPLE</td>
<td>Opr</td>
<td>10.6D</td>
<td>Compare signed quadword less than or equal</td>
</tr>
<tr>
<td>CMPLT</td>
<td>Opr</td>
<td>10.4D</td>
<td>Compare signed quadword less than</td>
</tr>
<tr>
<td>CMPTEQ</td>
<td>F-P</td>
<td>16.0A5</td>
<td>Compare T_floating equal</td>
</tr>
<tr>
<td>CMPTLE</td>
<td>F-P</td>
<td>16.0A7</td>
<td>Compare T_floating less than or equal</td>
</tr>
<tr>
<td>CMPTLT</td>
<td>F-P</td>
<td>16.0A6</td>
<td>Compare T_floating less than</td>
</tr>
<tr>
<td>CMPTUN</td>
<td>F-P</td>
<td>16.0A4</td>
<td>Compare T_floating unordered</td>
</tr>
<tr>
<td>CMPULE</td>
<td>Opr</td>
<td>10.3D</td>
<td>Compare unsigned quadword less than or equal</td>
</tr>
<tr>
<td>CMPULT</td>
<td>Opr</td>
<td>10.1D</td>
<td>Compare unsigned quadword less than</td>
</tr>
<tr>
<td>CPYS</td>
<td>F-P</td>
<td>17.020</td>
<td>Copy sign</td>
</tr>
<tr>
<td>CPYSE</td>
<td>F-P</td>
<td>17.022</td>
<td>Copy sign and exponent</td>
</tr>
<tr>
<td>CPYSN</td>
<td>F-P</td>
<td>17.021</td>
<td>Copy sign negate</td>
</tr>
<tr>
<td>CTLZ</td>
<td>Opr</td>
<td>1C.32</td>
<td>Count leading zero</td>
</tr>
<tr>
<td>CTPPOP</td>
<td>Opr</td>
<td>1C.30</td>
<td>Count population</td>
</tr>
<tr>
<td>CTTTZ</td>
<td>Opr</td>
<td>1C.33</td>
<td>Count trailing zero</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Format</td>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>CVTDG</td>
<td>F-P</td>
<td>15.09E</td>
<td>Convert (D_{\text{floating}}) to (G_{\text{floating}})</td>
</tr>
<tr>
<td>CVTGD</td>
<td>F-P</td>
<td>15.0AD</td>
<td>Convert (G_{\text{floating}}) to (D_{\text{floating}})</td>
</tr>
<tr>
<td>CVTGF</td>
<td>F-P</td>
<td>15.0AC</td>
<td>Convert (G_{\text{floating}}) to (F_{\text{floating}})</td>
</tr>
<tr>
<td>CVTQG</td>
<td>F-P</td>
<td>15.0AF</td>
<td>Convert (G_{\text{floating}}) to (quadword)</td>
</tr>
<tr>
<td>CVTLQ</td>
<td>F-P</td>
<td>17.010</td>
<td>Convert (longword) to (quadword)</td>
</tr>
<tr>
<td>CVTQF</td>
<td>F-P</td>
<td>15.0BC</td>
<td>Convert (quadword) to (F_{\text{floating}})</td>
</tr>
<tr>
<td>CVTQG</td>
<td>F-P</td>
<td>15.0BE</td>
<td>Convert (quadword) to (G_{\text{floating}})</td>
</tr>
<tr>
<td>CVTQL</td>
<td>F-P</td>
<td>17.030</td>
<td>Convert (quadword) to (longword)</td>
</tr>
<tr>
<td>CVTQS</td>
<td>F-P</td>
<td>16.0BC</td>
<td>Convert (quadword) to (S_{\text{floating}})</td>
</tr>
<tr>
<td>CVTQT</td>
<td>F-P</td>
<td>16.0BE</td>
<td>Convert (quadword) to (T_{\text{floating}})</td>
</tr>
<tr>
<td>CVTST</td>
<td>F-P</td>
<td>16.2AC</td>
<td>Convert (S_{\text{floating}}) to (T_{\text{floating}})</td>
</tr>
<tr>
<td>CVTTQ</td>
<td>F-P</td>
<td>16.0AF</td>
<td>Convert (T_{\text{floating}}) to (quadword)</td>
</tr>
<tr>
<td>CVTTST</td>
<td>F-P</td>
<td>16.0AC</td>
<td>Convert (T_{\text{floating}}) to (S_{\text{floating}})</td>
</tr>
<tr>
<td>DIVF</td>
<td>F-P</td>
<td>15.083</td>
<td>Divide (F_{\text{floating}})</td>
</tr>
<tr>
<td>DIVG</td>
<td>F-P</td>
<td>15.0A3</td>
<td>Divide (G_{\text{floating}})</td>
</tr>
<tr>
<td>DIVS</td>
<td>F-P</td>
<td>16.083</td>
<td>Divide (S_{\text{floating}})</td>
</tr>
<tr>
<td>DIVT</td>
<td>F-P</td>
<td>16.0A3</td>
<td>Divide (T_{\text{floating}})</td>
</tr>
<tr>
<td>ECB</td>
<td>Mfc</td>
<td>18.E800</td>
<td>Evict cache block</td>
</tr>
<tr>
<td>EQV</td>
<td>Opr</td>
<td>11.48</td>
<td>Logical equivalence</td>
</tr>
<tr>
<td>EXCB</td>
<td>Mfc</td>
<td>18.0400</td>
<td>Exception barrier</td>
</tr>
<tr>
<td>EXTBL</td>
<td>Opr</td>
<td>12.06</td>
<td>Extract byte low</td>
</tr>
<tr>
<td>EXTLH</td>
<td>Opr</td>
<td>12.6A</td>
<td>Extract longword high</td>
</tr>
<tr>
<td>EXTLL</td>
<td>Opr</td>
<td>12.26</td>
<td>Extract longword low</td>
</tr>
<tr>
<td>EXTQH</td>
<td>Opr</td>
<td>12.7A</td>
<td>Extract quadword high</td>
</tr>
<tr>
<td>EXTQL</td>
<td>Opr</td>
<td>12.36</td>
<td>Extract quadword low</td>
</tr>
<tr>
<td>EXTWH</td>
<td>Opr</td>
<td>12.5A</td>
<td>Extract word high</td>
</tr>
<tr>
<td>EXTWL</td>
<td>Opr</td>
<td>12.16</td>
<td>Extract word low</td>
</tr>
<tr>
<td>FBEQ</td>
<td>Bra</td>
<td>31</td>
<td>Floating branch (\text{if } = \text{zero})</td>
</tr>
<tr>
<td>FBGE</td>
<td>Bra</td>
<td>36</td>
<td>Floating branch (\text{if } \geq \text{zero})</td>
</tr>
<tr>
<td>FBGT</td>
<td>Bra</td>
<td>37</td>
<td>Floating branch (\text{if } &gt; \text{zero})</td>
</tr>
<tr>
<td>FBLE</td>
<td>Bra</td>
<td>33</td>
<td>Floating branch (\text{if } \leq \text{zero})</td>
</tr>
<tr>
<td>FBLT</td>
<td>Bra</td>
<td>32</td>
<td>Floating branch (\text{if } &lt; \text{zero})</td>
</tr>
<tr>
<td>FBNE</td>
<td>Bra</td>
<td>35</td>
<td>Floating branch (\text{if } \neq \text{zero})</td>
</tr>
<tr>
<td>FCMOVEQ</td>
<td>F-P</td>
<td>17.02A</td>
<td>FCMOVE if (= \text{zero})</td>
</tr>
<tr>
<td>FCMOVGE</td>
<td>F-P</td>
<td>17.02D</td>
<td>FCMOVE if (\geq \text{zero})</td>
</tr>
<tr>
<td>FCMOVGT</td>
<td>F-P</td>
<td>17.02F</td>
<td>FCMOVE if (&gt; \text{zero})</td>
</tr>
<tr>
<td>FCMOVLE</td>
<td>F-P</td>
<td>17.02E</td>
<td>FCMOVE if (\leq \text{zero})</td>
</tr>
<tr>
<td>FCMOVLT</td>
<td>F-P</td>
<td>17.02C</td>
<td>FCMOVE if (&lt; \text{zero})</td>
</tr>
<tr>
<td>FCMOVNE</td>
<td>F-P</td>
<td>17.02B</td>
<td>FCMOVE if (\neq \text{zero})</td>
</tr>
<tr>
<td>FETCH</td>
<td>Mfc</td>
<td>18.8000</td>
<td>Prefetch data</td>
</tr>
<tr>
<td>FETCH_M</td>
<td>Mfc</td>
<td>18.A000</td>
<td>Prefetch data, modify intent</td>
</tr>
<tr>
<td>FTOIS</td>
<td>F-P</td>
<td>1C.78</td>
<td>Floating to integer move, (S_{\text{floating}})</td>
</tr>
<tr>
<td>FTOIOT</td>
<td>F-P</td>
<td>1C.70</td>
<td>Floating to integer move, (T_{\text{floating}})</td>
</tr>
<tr>
<td>IMPLVER</td>
<td>Opr</td>
<td>11.6C</td>
<td>Implementation version</td>
</tr>
<tr>
<td>INSBL</td>
<td>Opr</td>
<td>12.0B</td>
<td>Insert byte low</td>
</tr>
<tr>
<td>INSLH</td>
<td>Opr</td>
<td>12.67</td>
<td>Insert longword high</td>
</tr>
<tr>
<td>INSLL</td>
<td>Opr</td>
<td>12.2B</td>
<td>Insert longword low</td>
</tr>
</tbody>
</table>

**Instruction Summary**
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSQH</td>
<td>Opr</td>
<td>12.77</td>
<td>Insert quadword high</td>
</tr>
<tr>
<td>INSQL</td>
<td>Opr</td>
<td>12.3B</td>
<td>Insert quadword low</td>
</tr>
<tr>
<td>INSWH</td>
<td>Opr</td>
<td>12.57</td>
<td>Insert word high</td>
</tr>
<tr>
<td>INSWL</td>
<td>Opr</td>
<td>12.1B</td>
<td>Insert word low</td>
</tr>
<tr>
<td>ITOFF</td>
<td>F-P</td>
<td>14.014</td>
<td>Integer to floating move, F_floating</td>
</tr>
<tr>
<td>ITOFS</td>
<td>F-P</td>
<td>14.004</td>
<td>Integer to floating move, S_floating</td>
</tr>
<tr>
<td>ITOFT</td>
<td>F-P</td>
<td>14.024</td>
<td>Integer to floating move, T_floating</td>
</tr>
<tr>
<td>JMP</td>
<td>Mbr</td>
<td>1A.0</td>
<td>Jump</td>
</tr>
<tr>
<td>JSR</td>
<td>Mbr</td>
<td>1A.1</td>
<td>Jump to subroutine</td>
</tr>
<tr>
<td>JSR_COROUTINE</td>
<td>Mbr</td>
<td>1A.3</td>
<td>Jump to subroutine return</td>
</tr>
<tr>
<td>LDA</td>
<td>Mem</td>
<td>08</td>
<td>Load address</td>
</tr>
<tr>
<td>LDAH</td>
<td>Mem</td>
<td>09</td>
<td>Load address high</td>
</tr>
<tr>
<td>LDBU</td>
<td>Mem</td>
<td>0A</td>
<td>Load zero-extended byte</td>
</tr>
<tr>
<td>LDWU</td>
<td>Mem</td>
<td>0C</td>
<td>Load zero-extended word</td>
</tr>
<tr>
<td>LDF</td>
<td>Mem</td>
<td>20</td>
<td>Load F_floating</td>
</tr>
<tr>
<td>LDG</td>
<td>Mem</td>
<td>21</td>
<td>Load G_floating</td>
</tr>
<tr>
<td>LDL</td>
<td>Mem</td>
<td>28</td>
<td>Load sign-extended longword</td>
</tr>
<tr>
<td>LDL_L</td>
<td>Mem</td>
<td>2A</td>
<td>Load sign-extended longword locked</td>
</tr>
<tr>
<td>LDQ</td>
<td>Mem</td>
<td>29</td>
<td>Load quadword</td>
</tr>
<tr>
<td>LDQ_L</td>
<td>Mem</td>
<td>2B</td>
<td>Load quadword locked</td>
</tr>
<tr>
<td>LDQ_U</td>
<td>Mem</td>
<td>08</td>
<td>Load unaligned quadword</td>
</tr>
<tr>
<td>LDS</td>
<td>Mem</td>
<td>22</td>
<td>Load S_floating</td>
</tr>
<tr>
<td>LTD</td>
<td>Mem</td>
<td>23</td>
<td>Load T_floating</td>
</tr>
<tr>
<td>MAXSB8</td>
<td>Opr</td>
<td>1C.3E</td>
<td>Vector signed byte maximum</td>
</tr>
<tr>
<td>MAXSW4</td>
<td>Opr</td>
<td>1C.3F</td>
<td>Vector signed word maximum</td>
</tr>
<tr>
<td>MAXUB8</td>
<td>Opr</td>
<td>1C.3C</td>
<td>Vector unsigned byte maximum</td>
</tr>
<tr>
<td>MAXUW4</td>
<td>Opr</td>
<td>1C.3D</td>
<td>Vector unsigned word maximum</td>
</tr>
<tr>
<td>MB</td>
<td>Mfc</td>
<td>18.4000</td>
<td>Memory barrier</td>
</tr>
<tr>
<td>MF_FPCR</td>
<td>F-P</td>
<td>17.025</td>
<td>Move from FPCR</td>
</tr>
<tr>
<td>MINSB8</td>
<td>Opr</td>
<td>1C.3B</td>
<td>Vector signed byte minimum</td>
</tr>
<tr>
<td>MINSW4</td>
<td>Opr</td>
<td>1C.39</td>
<td>Vector signed word minimum</td>
</tr>
<tr>
<td>MINUB8</td>
<td>Opr</td>
<td>1C.3A</td>
<td>Vector unsigned byte minimum</td>
</tr>
<tr>
<td>MINUW4</td>
<td>Opr</td>
<td>1C.3B</td>
<td>Vector unsigned word minimum</td>
</tr>
<tr>
<td>MSKBL</td>
<td>Opr</td>
<td>12.02</td>
<td>Mask byte low</td>
</tr>
<tr>
<td>MSKLL</td>
<td>Opr</td>
<td>12.62</td>
<td>Mask longword high</td>
</tr>
<tr>
<td>MSKQH</td>
<td>Opr</td>
<td>12.22</td>
<td>Mask longword low</td>
</tr>
<tr>
<td>MSKQL</td>
<td>Opr</td>
<td>12.72</td>
<td>Mask quadword high</td>
</tr>
<tr>
<td>MSKWH</td>
<td>Opr</td>
<td>12.52</td>
<td>Mask word high</td>
</tr>
<tr>
<td>MSKWL</td>
<td>Opr</td>
<td>12.12</td>
<td>Mask word low</td>
</tr>
<tr>
<td>MT_FPCR</td>
<td>F-P</td>
<td>17.024</td>
<td>Move to FPCR</td>
</tr>
<tr>
<td>MULF</td>
<td>F-P</td>
<td>15.082</td>
<td>Multiply F_floating</td>
</tr>
<tr>
<td>MULG</td>
<td>F-P</td>
<td>15.0A2</td>
<td>Multiply G_floating</td>
</tr>
<tr>
<td>MULL</td>
<td>Opr</td>
<td>13.00</td>
<td>Multiply longword</td>
</tr>
<tr>
<td>MULL/V</td>
<td></td>
<td>13.40</td>
<td>Multiply quadword</td>
</tr>
<tr>
<td>MULQ</td>
<td>Opr</td>
<td>13.20</td>
<td>Multiply quadword</td>
</tr>
<tr>
<td>MULQ/V</td>
<td></td>
<td>13.60</td>
<td>Multiply quadword</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Format</td>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>MULS</td>
<td>F-P</td>
<td>16.082</td>
<td>Multiply $S_{\text{floating}}$</td>
</tr>
<tr>
<td>MULT</td>
<td>F-P</td>
<td>16.0A2</td>
<td>Multiply $T_{\text{floating}}$</td>
</tr>
<tr>
<td>ORNOT</td>
<td>Opr</td>
<td>11.28</td>
<td>Logical sum with complement</td>
</tr>
<tr>
<td>PERR</td>
<td>Opr</td>
<td>1C.31</td>
<td>Pixel error</td>
</tr>
<tr>
<td>PKLB</td>
<td>Opr</td>
<td>1C.37</td>
<td>Pack longwords to bytes</td>
</tr>
<tr>
<td>PKWB</td>
<td>Opr</td>
<td>1C.36</td>
<td>Pack words to bytes</td>
</tr>
<tr>
<td>RC</td>
<td>Mfc</td>
<td>18.E000</td>
<td>Read and clear</td>
</tr>
<tr>
<td>RET</td>
<td>Mbr</td>
<td>1A.2</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>RPCC</td>
<td>Mfc</td>
<td>18.C000</td>
<td>Read process cycle counter</td>
</tr>
<tr>
<td>RS</td>
<td>Mfc</td>
<td>18.F000</td>
<td>Read and set</td>
</tr>
<tr>
<td>S4ADDL</td>
<td>Opr</td>
<td>10.02</td>
<td>Scaled add longword by 4</td>
</tr>
<tr>
<td>S4ADDOQ</td>
<td>Opr</td>
<td>10.22</td>
<td>Scaled add quadword by 4</td>
</tr>
<tr>
<td>S4SUBL</td>
<td>Opr</td>
<td>10.0B</td>
<td>Scaled subtract longword by 4</td>
</tr>
<tr>
<td>S4SUBQ</td>
<td>Opr</td>
<td>10.2B</td>
<td>Scaled subtract quadword by 4</td>
</tr>
<tr>
<td>S8ADDL</td>
<td>Opr</td>
<td>10.12</td>
<td>Scaled add longword by 8</td>
</tr>
<tr>
<td>S8ADDOQ</td>
<td>Opr</td>
<td>10.32</td>
<td>Scaled add quadword by 8</td>
</tr>
<tr>
<td>S8SUBL</td>
<td>Opr</td>
<td>10.1B</td>
<td>Scaled subtract longword by 8</td>
</tr>
<tr>
<td>S8SUBQ</td>
<td>Opr</td>
<td>10.3B</td>
<td>Scaled subtract quadword by 8</td>
</tr>
<tr>
<td>SEXTB</td>
<td>Opr</td>
<td>1C.00</td>
<td>Sign extend byte</td>
</tr>
<tr>
<td>SEXTW</td>
<td>Opr</td>
<td>1C.01</td>
<td>Sign extend word</td>
</tr>
<tr>
<td>SLL</td>
<td>Opr</td>
<td>12.39</td>
<td>Shift left logical</td>
</tr>
<tr>
<td>SQRTF</td>
<td>F-P</td>
<td>14.08A</td>
<td>Square root $F_{\text{floating}}$</td>
</tr>
<tr>
<td>SQRTG</td>
<td>F-P</td>
<td>14.0AA</td>
<td>Square root $G_{\text{floating}}$</td>
</tr>
<tr>
<td>SQRTS</td>
<td>F-P</td>
<td>14.0BB</td>
<td>Square root $S_{\text{floating}}$</td>
</tr>
<tr>
<td>SQRTT</td>
<td>F-P</td>
<td>14.0AB</td>
<td>Square root $T_{\text{floating}}$</td>
</tr>
<tr>
<td>SRA</td>
<td>Opr</td>
<td>12.3C</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>SRL</td>
<td>Opr</td>
<td>12.34</td>
<td>Shift right logical</td>
</tr>
<tr>
<td>STB</td>
<td>Mem</td>
<td>0E</td>
<td>Store byte</td>
</tr>
<tr>
<td>STF</td>
<td>Mem</td>
<td>24</td>
<td>Store $F_{\text{floating}}$</td>
</tr>
<tr>
<td>STG</td>
<td>Mem</td>
<td>25</td>
<td>Store $G_{\text{floating}}$</td>
</tr>
<tr>
<td>STS</td>
<td>Mem</td>
<td>26</td>
<td>Store $S_{\text{floating}}$</td>
</tr>
<tr>
<td>STL</td>
<td>Mem</td>
<td>2C</td>
<td>Store longword</td>
</tr>
<tr>
<td>STL_C</td>
<td>Mem</td>
<td>2E</td>
<td>Store longword conditional</td>
</tr>
<tr>
<td>STQ</td>
<td>Mem</td>
<td>2D</td>
<td>Store quadword</td>
</tr>
<tr>
<td>STQ_C</td>
<td>Mem</td>
<td>2F</td>
<td>Store quadword conditional</td>
</tr>
<tr>
<td>STU</td>
<td>Mem</td>
<td>0F</td>
<td>Store unaligned quadword</td>
</tr>
<tr>
<td>STT</td>
<td>Mem</td>
<td>27</td>
<td>Store $T_{\text{floating}}$</td>
</tr>
<tr>
<td>STW</td>
<td>Mem</td>
<td>0D</td>
<td>Store word</td>
</tr>
<tr>
<td>SUBF</td>
<td>F-P</td>
<td>15.081</td>
<td>Subtract $F_{\text{floating}}$</td>
</tr>
<tr>
<td>SUBG</td>
<td>F-P</td>
<td>15.0A1</td>
<td>Subtract $G_{\text{floating}}$</td>
</tr>
<tr>
<td>SUBL</td>
<td>Opr</td>
<td>10.09</td>
<td>Subtract longword</td>
</tr>
<tr>
<td>SUBL/V</td>
<td>Opr</td>
<td>10.49</td>
<td></td>
</tr>
<tr>
<td>SUBQ</td>
<td>Opr</td>
<td>10.29</td>
<td>Subtract quadword</td>
</tr>
<tr>
<td>SUBQ/V</td>
<td></td>
<td>10.69</td>
<td></td>
</tr>
<tr>
<td>SUBS</td>
<td>F-P</td>
<td>16.081</td>
<td>Subtract $S_{\text{floating}}$</td>
</tr>
<tr>
<td>SUBT</td>
<td>F-P</td>
<td>16.0A1</td>
<td>Subtract $T_{\text{floating}}$</td>
</tr>
<tr>
<td>TRAPB</td>
<td>Mfc</td>
<td>18.0000</td>
<td>Trap barrier</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Format</td>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>--------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>UMULH</td>
<td>Opr</td>
<td>13.30</td>
<td>Unsigned multiply quadword high</td>
</tr>
<tr>
<td>UNPKBL</td>
<td>Opr</td>
<td>1C.35</td>
<td>Unpack bytes to longwords</td>
</tr>
<tr>
<td>UNPKBW</td>
<td>Opr</td>
<td>1C.34</td>
<td>Unpack bytes to words</td>
</tr>
<tr>
<td>WH64</td>
<td>Mfc</td>
<td>18.F800</td>
<td>Write hint - 64 bytes</td>
</tr>
<tr>
<td>WMB</td>
<td>Mfc</td>
<td>18.4400</td>
<td>Write memory barrier</td>
</tr>
<tr>
<td>XOR</td>
<td>Opr</td>
<td>11.40</td>
<td>Logical difference</td>
</tr>
<tr>
<td>ZAP</td>
<td>Opr</td>
<td>12.30</td>
<td>Zero bytes</td>
</tr>
<tr>
<td>ZAPNOT</td>
<td>Opr</td>
<td>12.31</td>
<td>Zero bytes not</td>
</tr>
</tbody>
</table>
C.2 IEEE Floating-Point Instructions

Table C–3 lists the hexadecimal value of the 11-bit function code field for the IEEE floating-point instructions, with and without qualifiers. The opcode for the following instructions is 16\textsubscript{16}, except for SQRTS and SQRTT, which are opcode 14\textsubscript{16}.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>None</th>
<th>/C</th>
<th>/M</th>
<th>/D</th>
<th>/U</th>
<th>/UC</th>
<th>/UM</th>
<th>/UD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDS</td>
<td>080</td>
<td>000</td>
<td>040</td>
<td>0C0</td>
<td>180</td>
<td>100</td>
<td>140</td>
<td>1C0</td>
</tr>
<tr>
<td>ADDT</td>
<td>0A0</td>
<td>020</td>
<td>060</td>
<td>0E0</td>
<td>1A0</td>
<td>120</td>
<td>160</td>
<td>1E0</td>
</tr>
<tr>
<td>CMPTEQ</td>
<td>0A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPTLT</td>
<td>0A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPTLE</td>
<td>0A7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPTUN</td>
<td>0A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTQS</td>
<td>0BC</td>
<td>03C</td>
<td>07C</td>
<td>0FC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTQT</td>
<td>0BE</td>
<td>03E</td>
<td>07E</td>
<td>0FE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTTQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVS</td>
<td>083</td>
<td>003</td>
<td>043</td>
<td>0C3</td>
<td>183</td>
<td>103</td>
<td>143</td>
<td>1C3</td>
</tr>
<tr>
<td>DIVT</td>
<td>0A3</td>
<td>023</td>
<td>063</td>
<td>0E3</td>
<td>1A3</td>
<td>123</td>
<td>163</td>
<td>1E3</td>
</tr>
<tr>
<td>MULS</td>
<td>082</td>
<td>002</td>
<td>042</td>
<td>0C2</td>
<td>182</td>
<td>102</td>
<td>142</td>
<td>1C2</td>
</tr>
<tr>
<td>MULT</td>
<td>0A2</td>
<td>022</td>
<td>062</td>
<td>0E2</td>
<td>1A2</td>
<td>122</td>
<td>162</td>
<td>1E2</td>
</tr>
<tr>
<td>SQRTS</td>
<td>08B</td>
<td>00B</td>
<td>04B</td>
<td>0CB</td>
<td>18B</td>
<td>10B</td>
<td>14B</td>
<td>1CB</td>
</tr>
<tr>
<td>SQRTT</td>
<td>0AB</td>
<td>02B</td>
<td>06B</td>
<td>0EB</td>
<td>1AB</td>
<td>12B</td>
<td>16B</td>
<td>1EB</td>
</tr>
<tr>
<td>SUBS</td>
<td>081</td>
<td>001</td>
<td>041</td>
<td>0C1</td>
<td>181</td>
<td>101</td>
<td>141</td>
<td>1C1</td>
</tr>
<tr>
<td>SUBT</td>
<td>0A1</td>
<td>021</td>
<td>061</td>
<td>0E1</td>
<td>1A1</td>
<td>121</td>
<td>161</td>
<td>1E1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>/SU</th>
<th>/SUC</th>
<th>/SUM</th>
<th>/SUD</th>
<th>/SUI</th>
<th>/SUIC</th>
<th>/SUIM</th>
<th>/SUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDS</td>
<td>580</td>
<td>500</td>
<td>540</td>
<td>5C0</td>
<td>780</td>
<td>700</td>
<td>740</td>
<td>7C0</td>
</tr>
<tr>
<td>ADDT</td>
<td>5A0</td>
<td>520</td>
<td>560</td>
<td>5E0</td>
<td>7A0</td>
<td>720</td>
<td>760</td>
<td>7E0</td>
</tr>
<tr>
<td>CMPTEQ</td>
<td>5A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPTLT</td>
<td>5A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPTLE</td>
<td>5A7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPTUN</td>
<td>5A4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTQS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7BC</td>
<td>73C</td>
<td>77C</td>
</tr>
<tr>
<td>CVTQT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7BE</td>
<td>73E</td>
<td>77E</td>
</tr>
<tr>
<td>CVTTTS</td>
<td>5AC</td>
<td>52C</td>
<td>56C</td>
<td>5EC</td>
<td>7AC</td>
<td>72C</td>
<td>76C</td>
<td>7EC</td>
</tr>
<tr>
<td>DIVS</td>
<td>583</td>
<td>503</td>
<td>543</td>
<td>5C3</td>
<td>783</td>
<td>703</td>
<td>743</td>
<td>7C3</td>
</tr>
<tr>
<td>DIVT</td>
<td>5A3</td>
<td>523</td>
<td>563</td>
<td>5E3</td>
<td>7A3</td>
<td>723</td>
<td>763</td>
<td>7E3</td>
</tr>
<tr>
<td>MULS</td>
<td>582</td>
<td>502</td>
<td>542</td>
<td>5C2</td>
<td>782</td>
<td>702</td>
<td>742</td>
<td>7C2</td>
</tr>
<tr>
<td>MULT</td>
<td>5A2</td>
<td>522</td>
<td>562</td>
<td>5E2</td>
<td>7A2</td>
<td>722</td>
<td>762</td>
<td>7E2</td>
</tr>
<tr>
<td>SQRTS</td>
<td>58B</td>
<td>50B</td>
<td>54B</td>
<td>5CB</td>
<td>78B</td>
<td>70B</td>
<td>74B</td>
<td>7CB</td>
</tr>
<tr>
<td>SQRTT</td>
<td>5AB</td>
<td>52B</td>
<td>56B</td>
<td>5EB</td>
<td>7AB</td>
<td>72B</td>
<td>76B</td>
<td>7EB</td>
</tr>
<tr>
<td>SUBS</td>
<td>581</td>
<td>501</td>
<td>541</td>
<td>5C1</td>
<td>781</td>
<td>701</td>
<td>741</td>
<td>7C1</td>
</tr>
<tr>
<td>SUBT</td>
<td>5A1</td>
<td>521</td>
<td>561</td>
<td>5E1</td>
<td>7A1</td>
<td>721</td>
<td>761</td>
<td>7E1</td>
</tr>
</tbody>
</table>
### Table C–3 (Cont.): IEEE Floating-Point Instruction Function Codes

<table>
<thead>
<tr>
<th>None</th>
<th>/S</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTST</td>
<td>2AC  6AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>None</th>
<th>/C</th>
<th>/N</th>
<th>/VC</th>
<th>/SV</th>
<th>/SVC</th>
<th>/SVI</th>
<th>/SVIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTTQ</td>
<td>0AF  02F 1AF 12F 5AF 52F 7AF 72F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>/VD</th>
<th>/SVD</th>
<th>/SVID</th>
<th>/M</th>
<th>/VM</th>
<th>/SVM</th>
<th>/SVIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTTQ</td>
<td>0EF  1EF 5EF 7EF 06F 16F 56F 76F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Programming Note:**

In order to use CMPTxx with software completion trap handling, it is necessary to specify the /SU IEEE trap mode, even though an underflow trap is not possible.

In order to use CVTQS or CVTQT with software completion trap handling, it is necessary to specify the /SUI IEEE trap mode, even though an underflow trap is not possible.
C.3 VAX Floating-Point Instructions

Table C–4 lists the hexadecimal value of the 11-bit function code field for the VAX floating-point instructions. The opcode for the following instructions is $15_{16}$, except for SQRTF and SQRTG, which are opcode $14_{16}$.

Table C–4: VAX Floating-Point Instruction Function Codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>None</th>
<th>/C</th>
<th>/U</th>
<th>/UC</th>
<th>/S</th>
<th>/SC</th>
<th>/SU</th>
<th>/SUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDF</td>
<td>080</td>
<td>000</td>
<td>180</td>
<td>100</td>
<td>480</td>
<td>400</td>
<td>580</td>
<td>500</td>
</tr>
<tr>
<td>CVTDG</td>
<td>09E</td>
<td>01E</td>
<td>19E</td>
<td>11E</td>
<td>49E</td>
<td>41E</td>
<td>59E</td>
<td>51E</td>
</tr>
<tr>
<td>ADDG</td>
<td>0A0</td>
<td>020</td>
<td>1A0</td>
<td>120</td>
<td>4A0</td>
<td>420</td>
<td>5A0</td>
<td>520</td>
</tr>
<tr>
<td>CMPGEQ</td>
<td>0A5</td>
<td>4A5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPGLT</td>
<td>0A6</td>
<td></td>
<td>4A6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPGLE</td>
<td>0A7</td>
<td>4A7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTGF</td>
<td>0AC</td>
<td>02C</td>
<td>1AC</td>
<td>12C</td>
<td>4AC</td>
<td>42C</td>
<td>5AC</td>
<td>52C</td>
</tr>
<tr>
<td>CVTGD</td>
<td>0AD</td>
<td>02D</td>
<td>1AD</td>
<td>12D</td>
<td>4AD</td>
<td>42D</td>
<td>5AD</td>
<td>52D</td>
</tr>
<tr>
<td>CVTQG</td>
<td>See below</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTF</td>
<td>0BC</td>
<td>03C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTG</td>
<td>0BE</td>
<td>03E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVF</td>
<td>083</td>
<td>003</td>
<td>183</td>
<td>103</td>
<td>483</td>
<td>403</td>
<td>583</td>
<td>503</td>
</tr>
<tr>
<td>DIVG</td>
<td>0A3</td>
<td>023</td>
<td>1A3</td>
<td>123</td>
<td>4A3</td>
<td>423</td>
<td>5A3</td>
<td>523</td>
</tr>
<tr>
<td>MULF</td>
<td>082</td>
<td>002</td>
<td>182</td>
<td>102</td>
<td>482</td>
<td>402</td>
<td>582</td>
<td>502</td>
</tr>
<tr>
<td>MULG</td>
<td>0A2</td>
<td>022</td>
<td>1A2</td>
<td>122</td>
<td>4A2</td>
<td>422</td>
<td>5A2</td>
<td>522</td>
</tr>
<tr>
<td>SQRTF</td>
<td>08A</td>
<td>00A</td>
<td>18A</td>
<td>10A</td>
<td>48A</td>
<td>40A</td>
<td>58A</td>
<td>50A</td>
</tr>
<tr>
<td>SQRTG</td>
<td>0AA</td>
<td>02A</td>
<td>1AA</td>
<td>12A</td>
<td>4AA</td>
<td>42A</td>
<td>5AA</td>
<td>52A</td>
</tr>
<tr>
<td>SUBF</td>
<td>081</td>
<td>001</td>
<td>181</td>
<td>101</td>
<td>481</td>
<td>401</td>
<td>581</td>
<td>501</td>
</tr>
<tr>
<td>SUBG</td>
<td>0A1</td>
<td>021</td>
<td>1A1</td>
<td>121</td>
<td>4A1</td>
<td>421</td>
<td>5A1</td>
<td>521</td>
</tr>
</tbody>
</table>

C.4 Independent Floating-Point Instructions

Table C–5 lists the hexadecimal value of the 11-bit function code field for the floating-point instructions that are not directly tied to IEEE or VAX floating point. The opcode for the following instructions is $17_{16}$.

Table C–5: Independent Floating-Point Instruction Function Codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>None</th>
<th>/V</th>
<th>/VC</th>
<th>/S</th>
<th>/SC</th>
<th>/SV</th>
<th>/SVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTGQ</td>
<td>0AF</td>
<td>02F</td>
<td>1AF</td>
<td>12F</td>
<td>4AF</td>
<td>42F</td>
<td>5AF</td>
</tr>
</tbody>
</table>
Table C–5 (Cont.): Independent Floating-Point Instruction Function Codes

<table>
<thead>
<tr>
<th>None /V</th>
<th>/SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTQL</td>
<td>030</td>
</tr>
<tr>
<td>FCMOVEQ</td>
<td>02A</td>
</tr>
<tr>
<td>FCOMVGE</td>
<td>02D</td>
</tr>
<tr>
<td>FCOMVGT</td>
<td>02F</td>
</tr>
<tr>
<td>FCOMVLE</td>
<td>02E</td>
</tr>
<tr>
<td>FCOMVLT</td>
<td>02C</td>
</tr>
<tr>
<td>MF_FPCR</td>
<td>025</td>
</tr>
<tr>
<td>MT_FPCR</td>
<td>024</td>
</tr>
</tbody>
</table>
C.5 Opcode Summary

Table C–6 lists all Alpha opcodes from 00 (CALL_PAL) through 3F (BGT). In the table, the column headings that appear over the instructions have a granularity of $8_{16}$. The rows beneath the leftmost column supply the individual hex number to resolve that granularity.

If an instruction column has a 0 (zero) in the right (low) hex digit, replace that 0 with the number to the left of the backslash in the leftmost column on the instruction's row. If an instruction column has an 8 in the right (low) hexadecimal digit, replace that 8 with the number to the right of the backslash in the leftmost column.

For example, the third row (2/A) under the 10 column contains the symbol INTS*, representing all the integer shift instructions. The opcode for those instructions would then be $12_{16}$ because the 0 in 10 is replaced by the 2 in the leftmost column. Likewise, the third row under the 18 column contains the symbol JSR*, representing all jump instructions. The opcode for those instructions is $1A$ because the 8 in the heading is replaced by the number to the right of the backslash in the leftmost column.

The instruction format is listed under the instruction symbol.

The symbols in Table C–6 are explained in Table C–7.
## Table C–6: Opcode Summary

<table>
<thead>
<tr>
<th>00</th>
<th>08</th>
<th>10</th>
<th>18</th>
<th>20</th>
<th>28</th>
<th>30</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/8 PAL* (pal)</td>
<td>LDA (mem)</td>
<td>INTA* (op)</td>
<td>MISC* (mem)</td>
<td>LDF (mem)</td>
<td>LDL (mem)</td>
<td>BR (br)</td>
<td>BLBC (br)</td>
</tr>
<tr>
<td>1/9 Res</td>
<td>LDAH (mem)</td>
<td>INTL* (op)</td>
<td>\ PAL\</td>
<td>LDG (mem)</td>
<td>LDQ (mem)</td>
<td>FBEQ (br)</td>
<td>BEQ (br)</td>
</tr>
<tr>
<td>2/A Res</td>
<td>Res Res</td>
<td>INTS* (op)</td>
<td>JSR* (op)</td>
<td>LDS (mem)</td>
<td>LDL_L (mem)</td>
<td>FBLT (br)</td>
<td>BLT (br)</td>
</tr>
<tr>
<td>3/B Res</td>
<td>LDQ_U (mem)</td>
<td>INTM* (op)</td>
<td>\ PAL\</td>
<td>LDT (mem)</td>
<td>LDQ_L (mem)</td>
<td>FBEQ (br)</td>
<td>BLE (br)</td>
</tr>
<tr>
<td>4/C Res</td>
<td>Res</td>
<td>ITFP*</td>
<td>FPTI*</td>
<td>STF (mem)</td>
<td>STL (mem)</td>
<td>BSR (br)</td>
<td>BLBS (br)</td>
</tr>
<tr>
<td>5/D Res</td>
<td>STW</td>
<td>FLTV* (op)</td>
<td>\ PAL\</td>
<td>STG (mem)</td>
<td>STQ (mem)</td>
<td>FBNE (br)</td>
<td>BNE (br)</td>
</tr>
<tr>
<td>6/E Res</td>
<td>STB</td>
<td>FLTI* (op)</td>
<td>\ PAL\</td>
<td>STS (mem)</td>
<td>STL_C (mem)</td>
<td>FBGE (br)</td>
<td>BGE (br)</td>
</tr>
<tr>
<td>7/F Res</td>
<td>STQ_U (mem)</td>
<td>FLTL* (op)</td>
<td>\ PAL\</td>
<td>STT (mem)</td>
<td>STQ_C (mem)</td>
<td>FBGT (br)</td>
<td>BGT (br)</td>
</tr>
</tbody>
</table>

## Table C–7: Key to Opcode Summary (Table C–6)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLTI*</td>
<td>IEEE floating-point instruction opcodes</td>
</tr>
<tr>
<td>FLTL*</td>
<td>Floating-point Operate instruction opcodes</td>
</tr>
<tr>
<td>FLTV*</td>
<td>VAX floating-point instruction opcodes</td>
</tr>
<tr>
<td>FPTI*</td>
<td>Floating-point to integer register move opcodes</td>
</tr>
<tr>
<td>INTA*</td>
<td>Integer arithmetic instruction opcodes</td>
</tr>
<tr>
<td>INTL*</td>
<td>Integer logical instruction opcodes</td>
</tr>
<tr>
<td>INTM*</td>
<td>Integer multiply instruction opcodes</td>
</tr>
<tr>
<td>INTS*</td>
<td>Integer shift instruction opcodes</td>
</tr>
<tr>
<td>ITFP*</td>
<td>Integer to floating point register move opcodes</td>
</tr>
<tr>
<td>JSR*</td>
<td>Jump instruction opcodes</td>
</tr>
<tr>
<td>MISC*</td>
<td>Miscellaneous instruction opcodes</td>
</tr>
<tr>
<td>PAL*</td>
<td>PAL code instruction (CALL_PAL) opcodes</td>
</tr>
<tr>
<td>\ PAL\</td>
<td>Reserved for PAL code</td>
</tr>
<tr>
<td>Res</td>
<td>Reserved for Digital</td>
</tr>
</tbody>
</table>
### C.6 Common Architecture Opcodes in Numerical Order

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CALL_PAL</td>
<td>11.26</td>
</tr>
<tr>
<td>01</td>
<td>OPC01</td>
<td>11.28</td>
</tr>
<tr>
<td>02</td>
<td>OPC02</td>
<td>11.40</td>
</tr>
<tr>
<td>03</td>
<td>OPC03</td>
<td>11.44</td>
</tr>
<tr>
<td>04</td>
<td>OPC04</td>
<td>11.46</td>
</tr>
<tr>
<td>05</td>
<td>OPC05</td>
<td>11.48</td>
</tr>
<tr>
<td>06</td>
<td>OPC06</td>
<td>11.54</td>
</tr>
<tr>
<td>07</td>
<td>OPC07</td>
<td>11.64</td>
</tr>
<tr>
<td>08</td>
<td>LDA</td>
<td>11.66</td>
</tr>
<tr>
<td>09</td>
<td>LDAH</td>
<td>11.6C</td>
</tr>
<tr>
<td>0A</td>
<td>LD8U</td>
<td>12.02</td>
</tr>
<tr>
<td>0B</td>
<td>LDQ_U</td>
<td>12.06</td>
</tr>
<tr>
<td>0C</td>
<td>LDWU</td>
<td>12.0B</td>
</tr>
<tr>
<td>0D</td>
<td>STW</td>
<td>12.12</td>
</tr>
<tr>
<td>0E</td>
<td>STB</td>
<td>12.16</td>
</tr>
<tr>
<td>0F</td>
<td>STQ_U</td>
<td>12.1B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.00</td>
<td>ADDL</td>
<td>12.22</td>
</tr>
<tr>
<td>10.02</td>
<td>S4ADDL</td>
<td>12.26</td>
</tr>
<tr>
<td>10.09</td>
<td>SUBL</td>
<td>12.2B</td>
</tr>
<tr>
<td>10.08</td>
<td>S4SUBL</td>
<td>12.30</td>
</tr>
<tr>
<td>10.0F</td>
<td>CMPBGE</td>
<td>12.31</td>
</tr>
<tr>
<td>10.12</td>
<td>S8ADDL</td>
<td>12.32</td>
</tr>
<tr>
<td>10.1B</td>
<td>S8SUBL</td>
<td>12.34</td>
</tr>
<tr>
<td>10.1D</td>
<td>CMPULT</td>
<td>12.36</td>
</tr>
<tr>
<td>10.20</td>
<td>ADDQ</td>
<td>12.39</td>
</tr>
<tr>
<td>10.22</td>
<td>S4ADDQ</td>
<td>12.3B</td>
</tr>
<tr>
<td>10.29</td>
<td>SUBQ</td>
<td>12.3C</td>
</tr>
<tr>
<td>10.2B</td>
<td>S4SUBL</td>
<td>12.52</td>
</tr>
<tr>
<td>10.2D</td>
<td>CMPEQ</td>
<td>12.57</td>
</tr>
<tr>
<td>10.32</td>
<td>S8ADDQ</td>
<td>12.5A</td>
</tr>
<tr>
<td>10.3B</td>
<td>S8SUBL</td>
<td>12.62</td>
</tr>
<tr>
<td>10.3D</td>
<td>CMPULE</td>
<td>12.67</td>
</tr>
<tr>
<td>10.40</td>
<td>ADDL/V</td>
<td>12.6A</td>
</tr>
<tr>
<td>10.49</td>
<td>SUBL/V</td>
<td>12.72</td>
</tr>
<tr>
<td>10.4D</td>
<td>CMPLT</td>
<td>12.77</td>
</tr>
<tr>
<td>10.60</td>
<td>ADDQ/V</td>
<td>12.7A</td>
</tr>
<tr>
<td>10.69</td>
<td>SUBQ/V</td>
<td>13.00</td>
</tr>
<tr>
<td>10.6D</td>
<td>CMPLE</td>
<td>13.20</td>
</tr>
<tr>
<td>11.00</td>
<td>AND</td>
<td>13.30</td>
</tr>
<tr>
<td>11.08</td>
<td>BIC</td>
<td>13.40</td>
</tr>
<tr>
<td>11.14</td>
<td>CMOVVLBS</td>
<td>13.60</td>
</tr>
<tr>
<td>11.16</td>
<td>CMOVVLBC</td>
<td>14.00</td>
</tr>
<tr>
<td>11.20</td>
<td>BIS</td>
<td>14.00A</td>
</tr>
<tr>
<td>11.24</td>
<td>CMOVEQ</td>
<td>14.00B</td>
</tr>
</tbody>
</table>
Table C–8 (Cont.): Common Architecture Opcodes in Numerical Order

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.78B SQRTS/SUI</td>
<td>15.12F CVTGQ/VC</td>
<td>15.521 SUBG/SUC</td>
</tr>
<tr>
<td>14.7AB SQRTT/SUI</td>
<td>15.180 ADDF/U</td>
<td>15.522 MULG/SUC</td>
</tr>
<tr>
<td>14.7CB SQRTS/SUID</td>
<td>15.181 SUBF/U</td>
<td>15.523 DIVG/SUC</td>
</tr>
<tr>
<td>14.7EB SQRTT/SUID</td>
<td>15.182 MULF/U</td>
<td>15.52C CVTGF/SUC</td>
</tr>
<tr>
<td>15.000 ADDF/C</td>
<td>15.183 DIVF/U</td>
<td>15.52D CVTGD/SUC</td>
</tr>
<tr>
<td>15.001 SUBF/C</td>
<td>15.19E CVTDG/U</td>
<td>15.52F CVTGF/SVC</td>
</tr>
<tr>
<td>15.002 MULF/C</td>
<td>15.1A0 ADDG/U</td>
<td>15.580 ADDF/SU</td>
</tr>
<tr>
<td>15.003 DIVF/C</td>
<td>15.1A1 SUBG/U</td>
<td>15.581 SUBF/SU</td>
</tr>
<tr>
<td>15.01E CVTDG/C</td>
<td>15.1A2 MULG/U</td>
<td>15.582 MULF/SU</td>
</tr>
<tr>
<td>15.020 ADDG/C</td>
<td>15.1A3 DIVG/U</td>
<td>15.583 DIVF/SU</td>
</tr>
<tr>
<td>15.021 SUBG/C</td>
<td>15.1AC CVTGF/U</td>
<td>15.59E CVTDG/SU</td>
</tr>
<tr>
<td>15.022 MULG/C</td>
<td>15.1AD CVTGD/U</td>
<td>15.5A0 ADDG/SU</td>
</tr>
<tr>
<td>15.023 DIVG/C</td>
<td>15.1AF CVTGQ/V</td>
<td>15.5A1 SUBG/SU</td>
</tr>
<tr>
<td>15.02C CVTGF/C</td>
<td>15.400 ADDF/SC</td>
<td>15.5A2 MULG/SU</td>
</tr>
<tr>
<td>15.02D CVTGD/C</td>
<td>15.401 SUBF/SC</td>
<td>15.5A3 DIVG/SU</td>
</tr>
<tr>
<td>15.02F CVTQG/C</td>
<td>15.402 MULF/SC</td>
<td>15.5AC CVTGF/SU</td>
</tr>
<tr>
<td>15.03C CVTQF/C</td>
<td>15.403 DIVF/SC</td>
<td>15.5AD CVTGD/SU</td>
</tr>
<tr>
<td>15.03E CVTQG/C</td>
<td>15.41E CVTDG/SC</td>
<td>15.5AE CVTGQ/SU</td>
</tr>
<tr>
<td>15.080 ADDF</td>
<td>15.420 ADDG/SC</td>
<td>16.000 ADDS/C</td>
</tr>
<tr>
<td>15.081 SUBF</td>
<td>15.421 SUBG/SC</td>
<td>16.001 SUBS/C</td>
</tr>
<tr>
<td>15.082 MULF</td>
<td>15.422 MULG/SC</td>
<td>16.002 MULS/C</td>
</tr>
<tr>
<td>15.083 DIVF</td>
<td>15.423 DIVG/SC</td>
<td>16.003 DIVS/C</td>
</tr>
<tr>
<td>15.09E CVTDG</td>
<td>15.42C CVTGF/SC</td>
<td>16.020 ADT/SC</td>
</tr>
<tr>
<td>15.0A0 ADDG</td>
<td>15.42D CVTGD/SC</td>
<td>16.021 SUBT/C</td>
</tr>
<tr>
<td>15.0A1 SUBG</td>
<td>15.42F CVTQG/SC</td>
<td>16.022 MULT/C</td>
</tr>
<tr>
<td>15.0A2 MULG</td>
<td>15.480 ADDF/S</td>
<td>16.023 DIVT/C</td>
</tr>
<tr>
<td>15.0A3 DIVG</td>
<td>15.481 SUBF/S</td>
<td>16.02C CVTT/C</td>
</tr>
<tr>
<td>15.0A5 CMPGEQ</td>
<td>15.482 MULF/S</td>
<td>16.02E CVTGT/C</td>
</tr>
<tr>
<td>15.0A6 CMPGLT</td>
<td>15.483 DIVF/S</td>
<td>16.03C CVTQS/C</td>
</tr>
<tr>
<td>15.0A7 CMPGLE</td>
<td>15.49E CVTDG/S</td>
<td>16.03E CVTQ/C</td>
</tr>
<tr>
<td>15.0AC CVTGF</td>
<td>15.4A0 ADDG/S</td>
<td>16.040 ADDS/M</td>
</tr>
<tr>
<td>15.0AD CVTGD</td>
<td>15.4A1 SUBG/S</td>
<td>16.041 SUBS/M</td>
</tr>
<tr>
<td>15.0AF CVTQG</td>
<td>15.4A2 MULG/S</td>
<td>16.042 MULS/M</td>
</tr>
<tr>
<td>15.0BC CVTQF</td>
<td>15.4A3 DIVG/S</td>
<td>16.043 DIVS/M</td>
</tr>
<tr>
<td>15.0BE CVTQG</td>
<td>15.4A5 CMPGEQ/S</td>
<td>16.060 ADDT/M</td>
</tr>
<tr>
<td>15.100 ADDF/UC</td>
<td>15.4A6 CMPGLT/S</td>
<td>16.061 SUBT/M</td>
</tr>
<tr>
<td>15.101 SUBF/UC</td>
<td>15.4A7 CMPGLE/S</td>
<td>16.062 MULT/M</td>
</tr>
<tr>
<td>15.102 MULF/UC</td>
<td>15.4AC CVTGF/S</td>
<td>16.063 DIVT/M</td>
</tr>
<tr>
<td>15.103 DIVF/UC</td>
<td>15.4AD CVTDG/S</td>
<td>16.06C CVTT/M</td>
</tr>
<tr>
<td>15.11E CVTDG/UC</td>
<td>15.4AF CVTGF/S</td>
<td>16.06F CVTTQ/M</td>
</tr>
<tr>
<td>15.120 ADDF/UC</td>
<td>15.500 ADDF/SUC</td>
<td>16.07C CVTQS/M</td>
</tr>
<tr>
<td>15.121 SUBG/UC</td>
<td>15.501 SUBF/SUC</td>
<td>16.07E CVTQT/M</td>
</tr>
<tr>
<td>15.122 MULG/UC</td>
<td>15.502 MULF/SUC</td>
<td>16.080 ADDS</td>
</tr>
<tr>
<td>15.123 DIVG/UC</td>
<td>15.503 DIVF/SUC</td>
<td>16.081 SUBS</td>
</tr>
<tr>
<td>15.12C CVTGF/UC</td>
<td>15.51E CVTDG/SUC</td>
<td>16.082 MULS</td>
</tr>
<tr>
<td>15.12D CVTGD/UC</td>
<td>15.520 ADDG/SUC</td>
<td>16.083 DIVS</td>
</tr>
<tr>
<td>Opcode</td>
<td>Opcode</td>
<td>Opcode</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>16.0A0</td>
<td>ADDT</td>
<td>16.182</td>
</tr>
<tr>
<td>16.0A1</td>
<td>SUBT</td>
<td>16.183</td>
</tr>
<tr>
<td>16.0A2</td>
<td>MULT</td>
<td>16.1A0</td>
</tr>
<tr>
<td>16.0A3</td>
<td>DIVT</td>
<td>16.1A1</td>
</tr>
<tr>
<td>16.0A4</td>
<td>CMPTUN</td>
<td>16.1A2</td>
</tr>
<tr>
<td>16.0A5</td>
<td>CMPTEQ</td>
<td>16.1A3</td>
</tr>
<tr>
<td>16.0A6</td>
<td>CMPTLT</td>
<td>16.1A4</td>
</tr>
<tr>
<td>16.0A7</td>
<td>CMPTLE</td>
<td>16.1A5</td>
</tr>
<tr>
<td>16.0AC</td>
<td>CVTS</td>
<td>16.1A6</td>
</tr>
<tr>
<td>16.0AF</td>
<td>CVTTQ</td>
<td>16.1A7</td>
</tr>
<tr>
<td>16.0BC</td>
<td>CVTSQ</td>
<td>16.1AC</td>
</tr>
<tr>
<td>16.0BE</td>
<td>CVTQT</td>
<td>16.1C0</td>
</tr>
<tr>
<td>16.0C0</td>
<td>ADDS/D</td>
<td>16.1C1</td>
</tr>
<tr>
<td>16.0C1</td>
<td>SUBS/D</td>
<td>16.1C2</td>
</tr>
<tr>
<td>16.0C2</td>
<td>MULS/D</td>
<td>16.1C3</td>
</tr>
<tr>
<td>16.0C3</td>
<td>DIVS/D</td>
<td>16.1C4</td>
</tr>
<tr>
<td>16.0CE</td>
<td>DIVT/D</td>
<td>16.1C5</td>
</tr>
<tr>
<td>16.0CF</td>
<td>CVTTQ/D</td>
<td>16.1C6</td>
</tr>
<tr>
<td>16.0F0</td>
<td>CVTS/D</td>
<td>16.1C7</td>
</tr>
<tr>
<td>16.0F1</td>
<td>CVTTQ/D</td>
<td>16.1C8</td>
</tr>
<tr>
<td>16.0F2</td>
<td>CVTS/D</td>
<td>16.1C9</td>
</tr>
<tr>
<td>16.0F3</td>
<td>CVTTQ/D</td>
<td>16.1CA</td>
</tr>
<tr>
<td>16.0F4</td>
<td>CVTS/D</td>
<td>16.1CB</td>
</tr>
<tr>
<td>16.0F5</td>
<td>CVTTQ/D</td>
<td>16.1CC</td>
</tr>
<tr>
<td>16.0F6</td>
<td>CVTS/D</td>
<td>16.1CD</td>
</tr>
<tr>
<td>16.0F7</td>
<td>CVTTQ/D</td>
<td>16.1CE</td>
</tr>
<tr>
<td>16.0F8</td>
<td>CVTS/D</td>
<td>16.1CF</td>
</tr>
<tr>
<td>16.0F9</td>
<td>CVTTQ/D</td>
<td>16.1D0</td>
</tr>
<tr>
<td>16.0FA</td>
<td>CVTS/D</td>
<td>16.1D1</td>
</tr>
<tr>
<td>16.0FB</td>
<td>CVTTQ/D</td>
<td>16.1D2</td>
</tr>
<tr>
<td>16.0FC</td>
<td>CVTS/D</td>
<td>16.1D3</td>
</tr>
<tr>
<td>16.0FD</td>
<td>CVTTQ/D</td>
<td>16.1D4</td>
</tr>
<tr>
<td>16.0FE</td>
<td>CVTS/D</td>
<td>16.1D5</td>
</tr>
</tbody>
</table>

Instruction Summary  C–15
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Opcode</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.7A0</td>
<td>ADDT/SUI</td>
<td>18.4000</td>
</tr>
<tr>
<td>16.7A1</td>
<td>SUBT/SUI</td>
<td>18.4400</td>
</tr>
<tr>
<td>16.7A2</td>
<td>MULT/SUI</td>
<td>18.8000</td>
</tr>
<tr>
<td>16.7A3</td>
<td>DIVT/SUI</td>
<td>18.A000</td>
</tr>
<tr>
<td>16.7AC</td>
<td>CVTTS/SUI</td>
<td>18.C000</td>
</tr>
<tr>
<td>16.7AF</td>
<td>CVTTQ/SVI</td>
<td>18.E000</td>
</tr>
<tr>
<td>16.7BC</td>
<td>CVTQS/SUI</td>
<td>18.E800</td>
</tr>
<tr>
<td>16.7BE</td>
<td>CVTQT/SUI</td>
<td>18.F000</td>
</tr>
<tr>
<td>16.7C0</td>
<td>ADDS/SUID</td>
<td>18.F800</td>
</tr>
<tr>
<td>16.7C1</td>
<td>SUBS/SUID</td>
<td>19</td>
</tr>
<tr>
<td>16.7C2</td>
<td>MULS/SUID</td>
<td>1A.0</td>
</tr>
<tr>
<td>16.7C3</td>
<td>DIVS/SUID</td>
<td>1A.1</td>
</tr>
<tr>
<td>16.7E0</td>
<td>ADDT/SUID</td>
<td>1A.2</td>
</tr>
<tr>
<td>16.7E1</td>
<td>SUBT/SUID</td>
<td>1A.3</td>
</tr>
<tr>
<td>16.7E2</td>
<td>MULT/SUID</td>
<td>1B</td>
</tr>
<tr>
<td>16.7E3</td>
<td>DIVT/SUID</td>
<td>1C.00</td>
</tr>
<tr>
<td>16.7E4</td>
<td>CVTTS/SUID</td>
<td>1C.01</td>
</tr>
<tr>
<td>16.7E5</td>
<td>CVTTQ/SVID</td>
<td>1C.30</td>
</tr>
<tr>
<td>16.7E7</td>
<td>CVTQS/SUID</td>
<td>1C.31</td>
</tr>
<tr>
<td>16.7E8</td>
<td>CVTQT/SUID</td>
<td>1C.32</td>
</tr>
<tr>
<td>16.7E9</td>
<td>CVTQL</td>
<td>1C.33</td>
</tr>
<tr>
<td>16.7EA</td>
<td>CPYS</td>
<td>1C.34</td>
</tr>
<tr>
<td>16.7EB</td>
<td>CPYSN</td>
<td>1C.35</td>
</tr>
<tr>
<td>16.7EC</td>
<td>CPYSE</td>
<td>1C.36</td>
</tr>
<tr>
<td>16.7ED</td>
<td>MT_FPCR</td>
<td>1C.37</td>
</tr>
<tr>
<td>16.7EE</td>
<td>MF_FPCR</td>
<td>1C.38</td>
</tr>
<tr>
<td>16.7EF</td>
<td>FCMOVEQ</td>
<td>1C.39</td>
</tr>
<tr>
<td>16.7F0</td>
<td>FCMOVNE</td>
<td>1C.3A</td>
</tr>
<tr>
<td>16.7F1</td>
<td>FCMOVLT</td>
<td>1C.3B</td>
</tr>
<tr>
<td>16.7F2</td>
<td>FCMOVGE</td>
<td>1C.3C</td>
</tr>
<tr>
<td>16.7F3</td>
<td>FCMOVLE</td>
<td>1C.3D</td>
</tr>
<tr>
<td>16.7F4</td>
<td>FCMOVGT</td>
<td>1C.3E</td>
</tr>
<tr>
<td>16.7F5</td>
<td>CVTQL</td>
<td>1C.3F</td>
</tr>
<tr>
<td>16.7F6</td>
<td>CVTQLV</td>
<td>1C.70</td>
</tr>
<tr>
<td>16.7F7</td>
<td>CVTQL/SV</td>
<td>1C.78</td>
</tr>
<tr>
<td>18.0000</td>
<td>TRAPB</td>
<td>1D</td>
</tr>
<tr>
<td>18.0400</td>
<td>EXCB</td>
<td>1E</td>
</tr>
</tbody>
</table>
### C.7 OpenVMS Alpha PALcode Instruction Summary

#### Table C-9: OpenVMS Alpha Unprivileged PALcode Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMOVRM</td>
<td>00.00A1</td>
<td>Atomic move from register to memory</td>
</tr>
<tr>
<td>AMOVRR</td>
<td>00.00A0</td>
<td>Atomic move from register to register</td>
</tr>
<tr>
<td>BPT</td>
<td>00.0080</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>BUGCHK</td>
<td>00.0081</td>
<td>Bugcheck</td>
</tr>
<tr>
<td>CHMK</td>
<td>00.0083</td>
<td>Change mode to kernel</td>
</tr>
<tr>
<td>CHME</td>
<td>00.0082</td>
<td>Change mode to executive</td>
</tr>
<tr>
<td>CHMS</td>
<td>00.0084</td>
<td>Change mode to supervisor</td>
</tr>
<tr>
<td>CHMU</td>
<td>00.0085</td>
<td>Change mode to user</td>
</tr>
<tr>
<td>CLRFEN</td>
<td>00.00AE</td>
<td>Clear floating-point enable</td>
</tr>
<tr>
<td>GENTRAP</td>
<td>00.00AA</td>
<td>Generate software trap</td>
</tr>
<tr>
<td>IMB</td>
<td>00.0086</td>
<td>I-stream memory barrier</td>
</tr>
<tr>
<td>INSOSHIL</td>
<td>00.0087</td>
<td>Insert into longword queue at head interlocked</td>
</tr>
<tr>
<td>INSOHILR</td>
<td>00.00A2</td>
<td>Insert into longword queue at head interlocked resident</td>
</tr>
<tr>
<td>INSOHIQ</td>
<td>00.0089</td>
<td>Insert into quadword queue at head interlocked</td>
</tr>
<tr>
<td>INSOHIQR</td>
<td>00.00A4</td>
<td>Insert into quadword queue at head interlocked resident</td>
</tr>
<tr>
<td>INSOTIL</td>
<td>00.0088</td>
<td>Insert into longword queue at tail interlocked</td>
</tr>
<tr>
<td>INSQTIQ</td>
<td>00.008A</td>
<td>Insert into longword queue at tail interlocked</td>
</tr>
<tr>
<td>INSQTIQR</td>
<td>00.00A5</td>
<td>Insert into quadword queue at tail interlocked resident</td>
</tr>
<tr>
<td>INSQUEL</td>
<td>00.008B</td>
<td>Insert entry into longword queue</td>
</tr>
<tr>
<td>INSQUELD</td>
<td>00.008D</td>
<td>Insert entry into longword queue deferred</td>
</tr>
<tr>
<td>INSQUEQ</td>
<td>00.008C</td>
<td>Insert entry into quadword queue</td>
</tr>
<tr>
<td>INSQUEQD</td>
<td>00.008E</td>
<td>Insert entry into quadword queue deferred</td>
</tr>
<tr>
<td>PROBER</td>
<td>00.008F</td>
<td>Probe for read access</td>
</tr>
<tr>
<td>PROBEW</td>
<td>00.0090</td>
<td>Probe for write access</td>
</tr>
<tr>
<td>RD_PS</td>
<td>00.0091</td>
<td>Move processor status</td>
</tr>
<tr>
<td>READ_UNQ</td>
<td>00.009E</td>
<td>Read unique context</td>
</tr>
<tr>
<td>REI</td>
<td>00.0092</td>
<td>Return from exception or interrupt</td>
</tr>
<tr>
<td>REMQHIL</td>
<td>00.0093</td>
<td>Remove from longword queue at head interlocked</td>
</tr>
<tr>
<td>REMQHILR</td>
<td>00.00A6</td>
<td>Remove from longword queue at head interlocked resident</td>
</tr>
<tr>
<td>REMQHIQ</td>
<td>00.0095</td>
<td>Remove from quadword queue at head interlocked</td>
</tr>
<tr>
<td>REMQHIQR</td>
<td>00.00A8</td>
<td>Remove from quadword queue at head interlocked resident</td>
</tr>
<tr>
<td>REMQTIL</td>
<td>00.0094</td>
<td>Remove from longword queue at tail interlocked</td>
</tr>
<tr>
<td>REMQTILR</td>
<td>00.00A7</td>
<td>Remove from longword queue at tail interlocked resident</td>
</tr>
<tr>
<td>REMQTIQ</td>
<td>00.0096</td>
<td>Remove from quadword queue at tail interlocked</td>
</tr>
<tr>
<td>REMQTIQR</td>
<td>00.00A9</td>
<td>Remove from quadword queue at tail interlocked resident</td>
</tr>
<tr>
<td>REMQUEL</td>
<td>00.0097</td>
<td>Remove entry from longword queue</td>
</tr>
<tr>
<td>REMQUELD</td>
<td>00.0099</td>
<td>Remove entry from longword queue deferred</td>
</tr>
<tr>
<td>REMQUEQ</td>
<td>00.0098</td>
<td>Remove entry from quadword queue</td>
</tr>
<tr>
<td>REMQUEQD</td>
<td>00.009A</td>
<td>Remove entry from quadword queue deferred</td>
</tr>
<tr>
<td>RSCC</td>
<td>00.009D</td>
<td>Read system cycle counter</td>
</tr>
<tr>
<td>SWASTEN</td>
<td>00.009B</td>
<td>Swap AST enable for current mode</td>
</tr>
<tr>
<td>WRITE_UNQ</td>
<td>00.009F</td>
<td>Write unique context</td>
</tr>
<tr>
<td>WR_PS_SW</td>
<td>00.009C</td>
<td>Write processor status software field</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>CFLUSH</td>
<td>00.0001</td>
<td>Cache flush</td>
</tr>
<tr>
<td>CSERVE</td>
<td>00.0009</td>
<td>Console service</td>
</tr>
<tr>
<td>DRAINA</td>
<td>00.0002</td>
<td>Drain aborts</td>
</tr>
<tr>
<td>HALT</td>
<td>00.0000</td>
<td>Halt processor</td>
</tr>
<tr>
<td>LDOP</td>
<td>00.0003</td>
<td>Load quadword physical</td>
</tr>
<tr>
<td>MFPR_ASN</td>
<td>00.0006</td>
<td>Move from processor register ASN</td>
</tr>
<tr>
<td>MFPR_ESP</td>
<td>00.001E</td>
<td>Move from processor register ESP</td>
</tr>
<tr>
<td>MFPR_FEN</td>
<td>00.000B</td>
<td>Move from processor register FEN</td>
</tr>
<tr>
<td>MFPR_IPL</td>
<td>00.000E</td>
<td>Move from processor register IPL</td>
</tr>
<tr>
<td>MFPR_MCES</td>
<td>00.0010</td>
<td>Move from processor register MCES</td>
</tr>
<tr>
<td>MFPR_PCBB</td>
<td>00.0012</td>
<td>Move from processor register PCBB</td>
</tr>
<tr>
<td>MFPR_PRBR</td>
<td>00.0013</td>
<td>Move from processor register PRBR</td>
</tr>
<tr>
<td>MFPR_PTBR</td>
<td>00.0015</td>
<td>Move from processor register PTBR</td>
</tr>
<tr>
<td>MFPR_SCBB</td>
<td>00.0016</td>
<td>Move from processor register SCBB</td>
</tr>
<tr>
<td>MFPR_SISR</td>
<td>00.0019</td>
<td>Move from processor register SISR</td>
</tr>
<tr>
<td>MFPR_SSP</td>
<td>00.0020</td>
<td>Move from processor register SSP</td>
</tr>
<tr>
<td>MFPR_TBCHK</td>
<td>00.001A</td>
<td>Move from processor register TBCHK</td>
</tr>
<tr>
<td>MFPR_USP</td>
<td>00.0022</td>
<td>Move from processor register USP</td>
</tr>
<tr>
<td>MFPR_VPTB</td>
<td>00.0029</td>
<td>Move from processor register VPTB</td>
</tr>
<tr>
<td>MFPR_WHAMI</td>
<td>00.003F</td>
<td>Move from processor register WHAMI</td>
</tr>
<tr>
<td>MTPR_ASTEN</td>
<td>00.0026</td>
<td>Move to processor register ASTEN</td>
</tr>
<tr>
<td>MTPR_ASTSR</td>
<td>00.0027</td>
<td>Move to processor register ASTSR</td>
</tr>
<tr>
<td>MTPR_DATFX</td>
<td>00.002E</td>
<td>Move to processor register DATFX</td>
</tr>
<tr>
<td>MTPR_ESP</td>
<td>00.001F</td>
<td>Move to processor register ESP</td>
</tr>
<tr>
<td>MTPR_FEN</td>
<td>00.000B</td>
<td>Move to processor register FEN</td>
</tr>
<tr>
<td>MTPR_IPRI</td>
<td>00.000D</td>
<td>Move to processor register IPRI</td>
</tr>
<tr>
<td>MTPR_IPL</td>
<td>00.000E</td>
<td>Move to processor register IPL</td>
</tr>
<tr>
<td>MTPR_MCES</td>
<td>00.0011</td>
<td>Move to processor register MCES</td>
</tr>
<tr>
<td>MTPR_PERFMON</td>
<td>00.002B</td>
<td>Move to processor register PERFMON</td>
</tr>
<tr>
<td>MTPR_PRBR</td>
<td>00.0014</td>
<td>Move to processor register PRBR</td>
</tr>
<tr>
<td>MTPR_SCBB</td>
<td>00.0017</td>
<td>Move to processor register SCBB</td>
</tr>
<tr>
<td>MTPR_SIRR</td>
<td>00.0018</td>
<td>Move to processor register SIRR</td>
</tr>
<tr>
<td>MTPR_SSP</td>
<td>00.0021</td>
<td>Move to processor register SSP</td>
</tr>
<tr>
<td>MTPR_TBIA</td>
<td>00.001B</td>
<td>Move to processor register TBIA</td>
</tr>
<tr>
<td>MTPR_TBIAP</td>
<td>00.001C</td>
<td>Move to processor register TBIAP</td>
</tr>
<tr>
<td>MTPR_TBIISD</td>
<td>00.0024</td>
<td>Move to processor register TBIISD</td>
</tr>
<tr>
<td>MTPR_USB</td>
<td>00.0023</td>
<td>Move to processor register USP</td>
</tr>
<tr>
<td>MTPR_VPTB</td>
<td>00.002A</td>
<td>Move to processor register VPTB</td>
</tr>
<tr>
<td>STQP</td>
<td>00.0004</td>
<td>Store quadword physical</td>
</tr>
<tr>
<td>SWPCTX</td>
<td>00.0005</td>
<td>Swap privileged context</td>
</tr>
<tr>
<td>SWPPAL</td>
<td>00.000A</td>
<td>Swap PAL code image</td>
</tr>
<tr>
<td>WTIINT</td>
<td>00.003E</td>
<td>Wait for interrupt</td>
</tr>
</tbody>
</table>
### C.8 Digital UNIX PALcode Instruction Summary

#### Table C–11: Digital UNIX Unprivileged PALcode Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpt</td>
<td>00.0080</td>
<td>Breakpoint trap</td>
</tr>
<tr>
<td>bugchk</td>
<td>00.0081</td>
<td>Bugcheck</td>
</tr>
<tr>
<td>callsys</td>
<td>00.0083</td>
<td>System call</td>
</tr>
<tr>
<td>clrfen</td>
<td>00.00AE</td>
<td>Clear floating-point enable</td>
</tr>
<tr>
<td>gentrap</td>
<td>00.00AA</td>
<td>Generate software trap</td>
</tr>
<tr>
<td>imb</td>
<td>00.0086</td>
<td>I-stream memory barrier</td>
</tr>
<tr>
<td>rdunique</td>
<td>00.009E</td>
<td>Read unique value</td>
</tr>
<tr>
<td>urti</td>
<td>00.0092</td>
<td>Return from user mode trap</td>
</tr>
<tr>
<td>wrunique</td>
<td>00.009F</td>
<td>Write unique value</td>
</tr>
</tbody>
</table>

#### Table C–12: Digital UNIX Privileged PALcode Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cflush</td>
<td>00.0001</td>
<td>Cache flush</td>
</tr>
<tr>
<td>cserve</td>
<td>00.0009</td>
<td>Console service</td>
</tr>
<tr>
<td>draina</td>
<td>00.0002</td>
<td>Drain aborts</td>
</tr>
<tr>
<td>halt</td>
<td>00.0000</td>
<td>Halt the processor</td>
</tr>
<tr>
<td>rdmces</td>
<td>00.0010</td>
<td>Read machine check error summary register</td>
</tr>
<tr>
<td>rdps</td>
<td>00.0036</td>
<td>Read processor status</td>
</tr>
<tr>
<td>rdusp</td>
<td>00.003A</td>
<td>Read user stack pointer</td>
</tr>
<tr>
<td>rdval</td>
<td>00.0032</td>
<td>Read system value</td>
</tr>
<tr>
<td>retsys</td>
<td>00.003D</td>
<td>Return from system call</td>
</tr>
<tr>
<td>rti</td>
<td>00.003F</td>
<td>Return from trap or interrupt</td>
</tr>
<tr>
<td>swpctx</td>
<td>00.0030</td>
<td>Swap privileged context</td>
</tr>
<tr>
<td>swpipl</td>
<td>00.0035</td>
<td>Swap interrupt priority level</td>
</tr>
<tr>
<td>swppal</td>
<td>00.000A</td>
<td>Swap PALcode image</td>
</tr>
<tr>
<td>tbi</td>
<td>00.0033</td>
<td>Translation buffer invalidate</td>
</tr>
<tr>
<td>whami</td>
<td>00.003C</td>
<td>Who am I</td>
</tr>
<tr>
<td>wrnt</td>
<td>00.0034</td>
<td>Write system entry address</td>
</tr>
<tr>
<td>wrfen</td>
<td>00.002B</td>
<td>Write floating-point enable</td>
</tr>
<tr>
<td>wrfipir</td>
<td>00.000D</td>
<td>Write interprocessor interrupt request</td>
</tr>
<tr>
<td>wrkgp</td>
<td>00.0037</td>
<td>Write kernel global pointer</td>
</tr>
<tr>
<td>wrmces</td>
<td>00.0011</td>
<td>Write machine check error summary register</td>
</tr>
<tr>
<td>wrperfmon</td>
<td>00.0039</td>
<td>Performance monitoring function</td>
</tr>
<tr>
<td>wruspsp</td>
<td>00.0038</td>
<td>Write user stack pointer</td>
</tr>
<tr>
<td>wrval</td>
<td>00.0031</td>
<td>Write system value</td>
</tr>
<tr>
<td>wrvptptr</td>
<td>00.002D</td>
<td>Write virtual page table pointer</td>
</tr>
<tr>
<td>wtint</td>
<td>00.003E</td>
<td>Wait for interrupt</td>
</tr>
</tbody>
</table>
## C.9 Windows NT Alpha Instruction Summary

### Table C–13: Windows NT Alpha Unprivileged PALcode Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bpt</td>
<td>00.0080</td>
<td>Breakpoint trap</td>
</tr>
<tr>
<td>callkd</td>
<td>00.00AD</td>
<td>Call kernel debugger</td>
</tr>
<tr>
<td>callsys</td>
<td>00.0083</td>
<td>Call system service</td>
</tr>
<tr>
<td>gentrap</td>
<td>00.00AA</td>
<td>Generate trap</td>
</tr>
<tr>
<td>imb</td>
<td>00.0086</td>
<td>Instruction memory barrier</td>
</tr>
<tr>
<td>kbpt</td>
<td>00.00AC</td>
<td>Kernel breakpoint trap</td>
</tr>
<tr>
<td>rdtdeb</td>
<td>00.00AB</td>
<td>Read TEB internal processor register</td>
</tr>
</tbody>
</table>

### Table C–14: Windows NT Alpha Privileged PALcode instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csir</td>
<td>00.000D</td>
<td>Clear software interrupt request</td>
</tr>
<tr>
<td>di</td>
<td>00.0008</td>
<td>Disable interrupts</td>
</tr>
<tr>
<td>drain</td>
<td>00.0002</td>
<td>Drain aborts</td>
</tr>
<tr>
<td>dtbis</td>
<td>00.0016</td>
<td>Data translation buffer invalidate single</td>
</tr>
<tr>
<td>ei</td>
<td>00.0009</td>
<td>Enable interrupts</td>
</tr>
<tr>
<td>halt</td>
<td>00.0000</td>
<td>Trap to illegal instruction</td>
</tr>
<tr>
<td>initpal</td>
<td>00.0004</td>
<td>Initialize the PALcode</td>
</tr>
<tr>
<td>rdcounters</td>
<td>00.0030</td>
<td>Read PALcode event counters</td>
</tr>
<tr>
<td>rdirql</td>
<td>00.0007</td>
<td>Read current IRQL</td>
</tr>
<tr>
<td>rdmsp</td>
<td>00.0018</td>
<td>Read initial kernel stack</td>
</tr>
<tr>
<td>rdmc</td>
<td>00.0012</td>
<td>Read machine check error summary</td>
</tr>
<tr>
<td>rdpsr</td>
<td>00.001A</td>
<td>Read processor status register</td>
</tr>
<tr>
<td>rdstate</td>
<td>00.0031</td>
<td>Read internal processor state</td>
</tr>
<tr>
<td>rdthread</td>
<td>00.001E</td>
<td>Read the current thread value</td>
</tr>
<tr>
<td>reboot</td>
<td>00.0002</td>
<td>Transfer to console firmware</td>
</tr>
<tr>
<td>restart</td>
<td>00.0001</td>
<td>Restart the processor</td>
</tr>
<tr>
<td>retsys</td>
<td>00.000F</td>
<td>Return from system service call</td>
</tr>
<tr>
<td>rfe</td>
<td>00.000E</td>
<td>Return from exception</td>
</tr>
<tr>
<td>swpirql</td>
<td>00.0006</td>
<td>Swap IRQL</td>
</tr>
<tr>
<td>swpksp</td>
<td>00.0019</td>
<td>Swap initial kernel stack</td>
</tr>
<tr>
<td>swppal</td>
<td>00.000A</td>
<td>Swap PALcode</td>
</tr>
<tr>
<td>swpprocess</td>
<td>00.0011</td>
<td>Swap privileged process context</td>
</tr>
<tr>
<td>swpctx</td>
<td>00.0010</td>
<td>Swap privileged thread context</td>
</tr>
<tr>
<td>ssir</td>
<td>00.000C</td>
<td>Set software interrupt request</td>
</tr>
<tr>
<td>tbias</td>
<td>00.0014</td>
<td>Translation buffer invalidate all</td>
</tr>
<tr>
<td>tbis</td>
<td>00.0015</td>
<td>Translation buffer invalidate single</td>
</tr>
<tr>
<td>tbisasn</td>
<td>00.0017</td>
<td>Translation buffer invalidate single ASN</td>
</tr>
<tr>
<td>wrentry</td>
<td>00.0005</td>
<td>Write system entry</td>
</tr>
<tr>
<td>wrmces</td>
<td>00.0013</td>
<td>Write machine check error summary</td>
</tr>
<tr>
<td>wrperfmon</td>
<td>00.0020</td>
<td>Write performance monitoring values</td>
</tr>
</tbody>
</table>
Opcodes $00.0038_{16}$ through $00.003F_{16}$ are reserved for processor implementation-specific PAL code instructions. All other opcodes are reserved for use by Digital.
### Table C–15: PALcode Opcodes in Numerical Order

<table>
<thead>
<tr>
<th>Opcode16</th>
<th>Opcode10</th>
<th>OpenVMS Alpha</th>
<th>Digital UNIX</th>
<th>Windows NT Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>00.0000</td>
<td>00.0000</td>
<td>HALT</td>
<td>halt</td>
<td>halt</td>
</tr>
<tr>
<td>00.0001</td>
<td>00.0001</td>
<td>CFLUSH</td>
<td>cflush</td>
<td>restart</td>
</tr>
<tr>
<td>00.0002</td>
<td>00.0002</td>
<td>DRAIN A</td>
<td>drain A</td>
<td>drain A</td>
</tr>
<tr>
<td>00.0003</td>
<td>00.0003</td>
<td>LDQP</td>
<td>—</td>
<td>reboot</td>
</tr>
<tr>
<td>00.0004</td>
<td>00.0004</td>
<td>STQP</td>
<td>—</td>
<td>initpal</td>
</tr>
<tr>
<td>00.0005</td>
<td>00.0005</td>
<td>SWPCTX</td>
<td>—</td>
<td>wrentry</td>
</tr>
<tr>
<td>00.0006</td>
<td>00.0006</td>
<td>MFPR_ASN</td>
<td>—</td>
<td>swpirql</td>
</tr>
<tr>
<td>00.0007</td>
<td>00.0007</td>
<td>MTPR_ASTEN</td>
<td>—</td>
<td>rdirql</td>
</tr>
<tr>
<td>00.0008</td>
<td>00.0008</td>
<td>MTPR_ASTSR</td>
<td>—</td>
<td>di</td>
</tr>
<tr>
<td>00.0009</td>
<td>00.0009</td>
<td>CSERVE</td>
<td>cserve</td>
<td>e</td>
</tr>
<tr>
<td>00.000A</td>
<td>00.0010</td>
<td>SWPPAL</td>
<td>swppal</td>
<td>swppal</td>
</tr>
<tr>
<td>00.000B</td>
<td>00.0011</td>
<td>MFPR_FEN</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.000C</td>
<td>00.0012</td>
<td>MTPR_FEN</td>
<td>—</td>
<td>ssir</td>
</tr>
<tr>
<td>00.000D</td>
<td>00.0013</td>
<td>MTPR_IPR</td>
<td>wrpir</td>
<td>csir</td>
</tr>
<tr>
<td>00.000E</td>
<td>00.0014</td>
<td>MFPR_IPL</td>
<td>—</td>
<td>rfe</td>
</tr>
<tr>
<td>00.000F</td>
<td>00.0015</td>
<td>MTPR_IPL</td>
<td>—</td>
<td>retsys</td>
</tr>
<tr>
<td>00.0010</td>
<td>00.0016</td>
<td>MFPR_MCES</td>
<td>rdmc es</td>
<td>swpctx</td>
</tr>
<tr>
<td>00.0011</td>
<td>00.0017</td>
<td>MTPR_MCES</td>
<td>wrmc es</td>
<td>swpprocess</td>
</tr>
<tr>
<td>00.0012</td>
<td>00.0018</td>
<td>MFPR_PCBB</td>
<td>—</td>
<td>rdmc es</td>
</tr>
<tr>
<td>00.0013</td>
<td>00.0019</td>
<td>MFPR_PRBR</td>
<td>—</td>
<td>wrmc es</td>
</tr>
<tr>
<td>00.0014</td>
<td>00.0020</td>
<td>MTPR_PRBR</td>
<td>—</td>
<td>tbia</td>
</tr>
<tr>
<td>00.0015</td>
<td>00.0021</td>
<td>MFPR_PTBR</td>
<td>—</td>
<td>tbs</td>
</tr>
<tr>
<td>00.0016</td>
<td>00.0022</td>
<td>MFPR_SCBB</td>
<td>—</td>
<td>dtb</td>
</tr>
<tr>
<td>00.0017</td>
<td>00.0023</td>
<td>MTPR_SCBB</td>
<td>—</td>
<td>tbsas n</td>
</tr>
<tr>
<td>00.0018</td>
<td>00.0024</td>
<td>MTPR_SIRR</td>
<td>—</td>
<td>rdksp</td>
</tr>
<tr>
<td>00.0019</td>
<td>00.0025</td>
<td>MFPR_SISR</td>
<td>—</td>
<td>swpksp</td>
</tr>
<tr>
<td>00.001A</td>
<td>00.0026</td>
<td>MFPR_TBCHK</td>
<td>—</td>
<td>rdp sr</td>
</tr>
<tr>
<td>00.001B</td>
<td>00.0027</td>
<td>MTPR_TBIA</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.001C</td>
<td>00.0028</td>
<td>MTPR_TBIAP</td>
<td>—</td>
<td>rdp cr</td>
</tr>
<tr>
<td>00.001D</td>
<td>00.0029</td>
<td>MTPR_TBIS</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.001E</td>
<td>00.0030</td>
<td>MFPR_ESP</td>
<td>—</td>
<td>rdthread</td>
</tr>
<tr>
<td>00.001F</td>
<td>00.0031</td>
<td>MTPR_ESP</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0020</td>
<td>00.0032</td>
<td>MFPR_SSP</td>
<td>—</td>
<td>wrperfmon</td>
</tr>
<tr>
<td>00.0021</td>
<td>00.0033</td>
<td>MTPR_SSP</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0022</td>
<td>00.0034</td>
<td>MFPR_USP</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0023</td>
<td>00.0035</td>
<td>MTPR-USP</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0024</td>
<td>00.0036</td>
<td>MTPR_TBISD</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0025</td>
<td>00.0037</td>
<td>MTPR_TBISI</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0026</td>
<td>00.0038</td>
<td>MFPR_ASTEN</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0027</td>
<td>00.0039</td>
<td>MFPR_ASTSR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0029</td>
<td>00.0041</td>
<td>MFPR_VPTB</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.002A</td>
<td>00.0042</td>
<td>MTPR_VPTB</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.002B</td>
<td>00.0043</td>
<td>MTPR_PERFMON</td>
<td>wrfen</td>
<td>—</td>
</tr>
<tr>
<td>00.002D</td>
<td>00.0045</td>
<td>—</td>
<td>wrvptptr</td>
<td>—</td>
</tr>
<tr>
<td>Opcode_{16}</td>
<td>Opcode_{10}</td>
<td>OpenVMS Alpha</td>
<td>Digital UNIX</td>
<td>Windows NT Alpha</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>---------------</td>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>00.002E</td>
<td>00.0046</td>
<td>MTPR_DATFX</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0030</td>
<td>00.0048</td>
<td>—</td>
<td>swpctx</td>
<td>rdcounters</td>
</tr>
<tr>
<td>00.0031</td>
<td>00.0049</td>
<td>—</td>
<td>wrval</td>
<td>rdstate</td>
</tr>
<tr>
<td>00.0032</td>
<td>00.0050</td>
<td>—</td>
<td>rdval</td>
<td>—</td>
</tr>
<tr>
<td>00.0033</td>
<td>00.0051</td>
<td>—</td>
<td>tbi</td>
<td>—</td>
</tr>
<tr>
<td>00.0034</td>
<td>00.0052</td>
<td>—</td>
<td>writ</td>
<td>—</td>
</tr>
<tr>
<td>00.0035</td>
<td>00.0053</td>
<td>—</td>
<td>swpipl</td>
<td>—</td>
</tr>
<tr>
<td>00.0036</td>
<td>00.0054</td>
<td>—</td>
<td>rdps</td>
<td>—</td>
</tr>
<tr>
<td>00.0037</td>
<td>00.0055</td>
<td>—</td>
<td>wrkgp</td>
<td>—</td>
</tr>
<tr>
<td>00.0038</td>
<td>00.0056</td>
<td>—</td>
<td>wrusp</td>
<td>—</td>
</tr>
<tr>
<td>00.0039</td>
<td>00.0057</td>
<td>—</td>
<td>wrperfmon</td>
<td>—</td>
</tr>
<tr>
<td>00.003A</td>
<td>00.0058</td>
<td>—</td>
<td>rdusp</td>
<td>—</td>
</tr>
<tr>
<td>00.003C</td>
<td>00.0060</td>
<td>—</td>
<td>whami</td>
<td>—</td>
</tr>
<tr>
<td>00.003D</td>
<td>00.0061</td>
<td>—</td>
<td>retsy</td>
<td>—</td>
</tr>
<tr>
<td>00.003E</td>
<td>00.0062</td>
<td>WTINT</td>
<td>—</td>
<td>wtint</td>
</tr>
<tr>
<td>00.003F</td>
<td>00.0063</td>
<td>MFPR_WHAMI</td>
<td>rti</td>
<td>—</td>
</tr>
<tr>
<td>00.0080</td>
<td>00.0128</td>
<td>BPT</td>
<td>bpt</td>
<td>bpt</td>
</tr>
<tr>
<td>00.0081</td>
<td>00.0129</td>
<td>BUGCHK</td>
<td>bugchk</td>
<td>—</td>
</tr>
<tr>
<td>00.0082</td>
<td>00.0130</td>
<td>CHME</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0083</td>
<td>00.0131</td>
<td>CHMK</td>
<td>callsys</td>
<td>callsys</td>
</tr>
<tr>
<td>00.0084</td>
<td>00.0132</td>
<td>CHMS</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0085</td>
<td>00.0133</td>
<td>CHMU</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0086</td>
<td>00.0134</td>
<td>IMB</td>
<td>imb</td>
<td>imb</td>
</tr>
<tr>
<td>00.0087</td>
<td>00.0135</td>
<td>INSQHIL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0088</td>
<td>00.0136</td>
<td>INSQTL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0089</td>
<td>00.0137</td>
<td>INSQHIQ</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.008A</td>
<td>00.0138</td>
<td>INSQTIQ</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.008B</td>
<td>00.0139</td>
<td>INSQUEL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.008C</td>
<td>00.0140</td>
<td>INSQUEQ</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.008D</td>
<td>00.0141</td>
<td>INSQUEL/D</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.008E</td>
<td>00.0142</td>
<td>INSQUEQ/D</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.008F</td>
<td>00.0143</td>
<td>PROBER</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0090</td>
<td>00.0144</td>
<td>PROBEW</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0091</td>
<td>00.0145</td>
<td>RD_PS</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0092</td>
<td>00.0146</td>
<td>REI</td>
<td>urti</td>
<td>—</td>
</tr>
<tr>
<td>00.0093</td>
<td>00.0147</td>
<td>REMQHIL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0094</td>
<td>00.0148</td>
<td>REMQTIL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0095</td>
<td>00.0149</td>
<td>REMQHIQ</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0096</td>
<td>00.0150</td>
<td>REMQTIQ</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0097</td>
<td>00.0151</td>
<td>REMQUEL</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0098</td>
<td>00.0152</td>
<td>REMQUEQ</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.0099</td>
<td>00.0153</td>
<td>REMQUEL/D</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.009A</td>
<td>00.0154</td>
<td>REMQUEQ/D</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.009B</td>
<td>00.0155</td>
<td>SWASTEN</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.009C</td>
<td>00.0156</td>
<td>WR_PS_SW</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.009D</td>
<td>00.0157</td>
<td>RSCC</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.009E</td>
<td>00.0158</td>
<td>READ_UNQ</td>
<td>rdunique</td>
<td>—</td>
</tr>
<tr>
<td>Opcode16</td>
<td>Opcode10</td>
<td>OpenVMS Alpha</td>
<td>Digital UNIX</td>
<td>Windows NT Alpha</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>---------------</td>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>00.009F</td>
<td>00.0159</td>
<td>WRITE_UNQ</td>
<td>wrunique</td>
<td>—</td>
</tr>
<tr>
<td>00.00A0</td>
<td>00.0160</td>
<td>AMOVRR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A1</td>
<td>00.0161</td>
<td>AMOVRM</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A2</td>
<td>00.0162</td>
<td>INSQHILR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A3</td>
<td>00.0163</td>
<td>INSQTLIR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A4</td>
<td>00.0164</td>
<td>INSQHIQR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A5</td>
<td>00.0165</td>
<td>INSQTIQR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A6</td>
<td>00.0166</td>
<td>REMQHILR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A7</td>
<td>00.0167</td>
<td>REMQTLIR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A8</td>
<td>00.0168</td>
<td>REMQHIQR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00A9</td>
<td>00.0169</td>
<td>REMQTIQR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>00.00AA</td>
<td>00.0170</td>
<td>GENTRAP</td>
<td>gentrap</td>
<td>gentrap</td>
</tr>
<tr>
<td>00.00AB</td>
<td>00.0171</td>
<td>—</td>
<td>—</td>
<td>rdteb</td>
</tr>
<tr>
<td>00.00AC</td>
<td>00.0172</td>
<td>—</td>
<td>—</td>
<td>kbpt</td>
</tr>
<tr>
<td>00.00AD</td>
<td>00.0173</td>
<td>—</td>
<td>—</td>
<td>callkd</td>
</tr>
<tr>
<td>00.00AE</td>
<td>00.0174</td>
<td>CLRFEN</td>
<td>drfen</td>
<td></td>
</tr>
</tbody>
</table>
C.11 Required PALcode Function Codes

The opcodes listed in Table C–16 are required for all Alpha implementations. The notation used is oo.ffff, where oo is the hexadecimal 6-bit opcode and ffff is the hexadecimal 26-bit function code.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Type</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAIN A</td>
<td>Privileged</td>
<td>00.0002</td>
</tr>
<tr>
<td>HALT</td>
<td>Privileged</td>
<td>00.0000</td>
</tr>
<tr>
<td>IMB</td>
<td>Unprivileged</td>
<td>00.0086</td>
</tr>
</tbody>
</table>

C.12 Opcodes Reserved to PALcode

The opcodes listed in Table C–17 are reserved for use in implementing PALcode.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Mnemonic</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAL19</td>
<td>19</td>
<td>PAL1B</td>
</tr>
<tr>
<td>PAL1E</td>
<td>1E</td>
<td>PAL1F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PAL1D</td>
</tr>
</tbody>
</table>

C.13 Opcodes Reserved to Digital

The opcodes listed in Table C–18 are reserved to Digital.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Mnemonic</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPC01</td>
<td>01</td>
<td>OPC02</td>
</tr>
<tr>
<td>OPC04</td>
<td>04</td>
<td>OPC05</td>
</tr>
<tr>
<td>OPC07</td>
<td>07</td>
<td>OPC03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OPC06</td>
</tr>
</tbody>
</table>

Programming Note:

The code points 18.4800 and 18.4C00 are reserved for adding weaker memory barrier instructions. Those code points must operate as a Memory Barrier instruction (MB 18.4000) for implementations that precede their definition as weaker memory barrier instructions. Software must use the 18.4000 code point for MB.
C.14 Unused Function Code Behavior

Unused function codes for all opcodes assigned (not reserved) in the Version 5 Alpha architecture specification (May 1992) produce UNPREDICTABLE but not UNDEFINED results; they are not security holes.

Unused function codes for opcodes defined as reserved in the Version 5 Alpha architecture specification produce an illegal instruction trap. Those opcodes are 01, 02, 03, 04, 05, 06, 07, 0A, 0C, 0D, 0E, 14, 19, 1B, 1C, 1D, 1E, and 1F. Unused function codes for those opcodes reserved to PALcode produce an illegal instruction trap only if not used in the PALcode environment.
C.15 ASCII Character Set

This section contains the 7-bit ASCII character set and the corresponding hexadecimal value for each character.

Table C–19: ASCII Character Set

<table>
<thead>
<tr>
<th>Char</th>
<th>Hex Code</th>
<th>Char</th>
<th>Hex Code</th>
<th>Char</th>
<th>Hex Code</th>
<th>Char</th>
<th>Hex Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>0</td>
<td>SP</td>
<td>20</td>
<td>@</td>
<td>40</td>
<td>'</td>
<td>60</td>
</tr>
<tr>
<td>SOH</td>
<td>1</td>
<td>!</td>
<td>21</td>
<td>A</td>
<td>41</td>
<td>a</td>
<td>61</td>
</tr>
<tr>
<td>STX</td>
<td>2</td>
<td>&quot;</td>
<td>22</td>
<td>B</td>
<td>42</td>
<td>b</td>
<td>62</td>
</tr>
<tr>
<td>ETX</td>
<td>3</td>
<td>#</td>
<td>23</td>
<td>C</td>
<td>43</td>
<td>c</td>
<td>63</td>
</tr>
<tr>
<td>EOT</td>
<td>4</td>
<td>$</td>
<td>24</td>
<td>D</td>
<td>44</td>
<td>d</td>
<td>64</td>
</tr>
<tr>
<td>ENQ</td>
<td>5</td>
<td>%</td>
<td>25</td>
<td>E</td>
<td>45</td>
<td>e</td>
<td>65</td>
</tr>
<tr>
<td>ACK</td>
<td>6</td>
<td>&amp;</td>
<td>26</td>
<td>F</td>
<td>46</td>
<td>f</td>
<td>66</td>
</tr>
<tr>
<td>BEL</td>
<td>7</td>
<td>'</td>
<td>27</td>
<td>G</td>
<td>47</td>
<td>g</td>
<td>67</td>
</tr>
<tr>
<td>BS</td>
<td>8</td>
<td>(</td>
<td>28</td>
<td>H</td>
<td>48</td>
<td>h</td>
<td>68</td>
</tr>
<tr>
<td>HT</td>
<td>9</td>
<td>)</td>
<td>29</td>
<td>I</td>
<td>49</td>
<td>i</td>
<td>69</td>
</tr>
<tr>
<td>LF</td>
<td>A</td>
<td>*</td>
<td>2A</td>
<td>J</td>
<td>4A</td>
<td>J</td>
<td>6A</td>
</tr>
<tr>
<td>VT</td>
<td>B</td>
<td>+</td>
<td>2B</td>
<td>K</td>
<td>4B</td>
<td>k</td>
<td>6B</td>
</tr>
<tr>
<td>FF</td>
<td>C</td>
<td>,</td>
<td>2C</td>
<td>L</td>
<td>4C</td>
<td>l</td>
<td>6C</td>
</tr>
<tr>
<td>CR</td>
<td>D</td>
<td>-</td>
<td>2D</td>
<td>M</td>
<td>4D</td>
<td>m</td>
<td>6D</td>
</tr>
<tr>
<td>SO</td>
<td>E</td>
<td>.</td>
<td>2E</td>
<td>N</td>
<td>4E</td>
<td>n</td>
<td>6E</td>
</tr>
<tr>
<td>SI</td>
<td>F</td>
<td>/</td>
<td>2F</td>
<td>O</td>
<td>4F</td>
<td>o</td>
<td>6F</td>
</tr>
<tr>
<td>DLE</td>
<td>10</td>
<td>0</td>
<td>30</td>
<td>P</td>
<td>50</td>
<td>p</td>
<td>70</td>
</tr>
<tr>
<td>DC1</td>
<td>11</td>
<td>1</td>
<td>31</td>
<td>Q</td>
<td>51</td>
<td>q</td>
<td>71</td>
</tr>
<tr>
<td>DC2</td>
<td>12</td>
<td>2</td>
<td>32</td>
<td>R</td>
<td>52</td>
<td>r</td>
<td>72</td>
</tr>
<tr>
<td>DC3</td>
<td>13</td>
<td>3</td>
<td>33</td>
<td>S</td>
<td>53</td>
<td>s</td>
<td>73</td>
</tr>
<tr>
<td>DC4</td>
<td>14</td>
<td>4</td>
<td>34</td>
<td>T</td>
<td>54</td>
<td>t</td>
<td>74</td>
</tr>
<tr>
<td>NAK</td>
<td>15</td>
<td>5</td>
<td>35</td>
<td>U</td>
<td>55</td>
<td>u</td>
<td>75</td>
</tr>
<tr>
<td>SYN</td>
<td>16</td>
<td>6</td>
<td>36</td>
<td>V</td>
<td>56</td>
<td>v</td>
<td>76</td>
</tr>
<tr>
<td>ETB</td>
<td>17</td>
<td>7</td>
<td>37</td>
<td>W</td>
<td>57</td>
<td>w</td>
<td>77</td>
</tr>
<tr>
<td>CAN</td>
<td>18</td>
<td>8</td>
<td>38</td>
<td>X</td>
<td>58</td>
<td>x</td>
<td>78</td>
</tr>
<tr>
<td>EM</td>
<td>19</td>
<td>9</td>
<td>39</td>
<td>Y</td>
<td>59</td>
<td>y</td>
<td>79</td>
</tr>
<tr>
<td>SUB</td>
<td>1A</td>
<td>:</td>
<td>3A</td>
<td>Z</td>
<td>5A</td>
<td>z</td>
<td>7A</td>
</tr>
<tr>
<td>ESC</td>
<td>1B</td>
<td>;</td>
<td>3B</td>
<td>[</td>
<td>5B</td>
<td>}</td>
<td>7B</td>
</tr>
<tr>
<td>FS</td>
<td>1C</td>
<td>&lt;</td>
<td>3C</td>
<td>\</td>
<td>5C</td>
<td>]</td>
<td>7C</td>
</tr>
<tr>
<td>GS</td>
<td>1D</td>
<td>=</td>
<td>3D</td>
<td>]</td>
<td>5D</td>
<td>}</td>
<td>7D</td>
</tr>
<tr>
<td>RS</td>
<td>1E</td>
<td>&gt;</td>
<td>3E</td>
<td>^</td>
<td>5E</td>
<td>~</td>
<td>7E</td>
</tr>
<tr>
<td>US</td>
<td>1F</td>
<td>?</td>
<td>3F</td>
<td>_</td>
<td>5F</td>
<td>DEL</td>
<td>7F</td>
</tr>
</tbody>
</table>
This appendix contains a table of the processor type assignments and the architecture mask (AMASK) and implementation value (IMPLVER) assignments.

## D.1 Processor Type Assignments

The following processor types are defined.

<table>
<thead>
<tr>
<th>Major Type</th>
<th>Minor Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EV3</td>
<td></td>
</tr>
</tbody>
</table>
| 2          | EV4 (21064)| 0 = Pass 2 or 2.1  
                  |            | 1 = Pass 3 (also EV4s) |
| 3          | Simulation |             |
| 4          | LCA Family: |             |
|            | LCA4s (21066)|             |
|            | LCA4s embedded (21068)|             |
|            | LCA45 (21066A, 21068A)|             |
|            | 0 = Reserved |             |
|            | 1 = Pass 1 or 1.1 (21066) |             |
|            | 2 = Pass 2 (21066) |             |
|            | 3 = Pass 1 or 1.1 (21068) |             |
|            | 4 = Pass 2 (21068) |             |
|            | 5 = Pass 1 (21066A) |             |
|            | 6 = Pass 1 (21068A) |             |
| 5          | EV5 (21164)| 0 = Reserved  
                  |            | 1 = Pass 2, 2.2 (rev BA, CA) |
|            |            | 2 = Pass 2.3 (rev DA, EA) |
|            |            | 3 = Pass 3 |
|            |            | 4 = Pass 3.2 |
|            |            | 5 = Pass 4 |
| 6          | EV45 (21064A)| 0 = Reserved  
                  |            | 1 = Pass 1 |
|            |            | 2 = Pass 1.1 |
|            |            | 3 = Pass 2 |
| 7          | EV56 (21164A)| 0 = Reserved  
                  |            | 1 = Pass 1 |
|            |            | 2 = Pass 2 |
| 8          | EV6 (21264)| 0 = Reserved  
                  |            | 1 = Pass 1 |
| 9          | PCA56 (21164PC)| 0 = Reserved |
Table D–1 (Cont.): Processor Type Assignments

<table>
<thead>
<tr>
<th>Major Type</th>
<th>Minor Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pass 1</td>
</tr>
</tbody>
</table>

For OpenVMS Alpha and Digital UNIX, the processor types are stored in the Per-CPU Slot Table (SLOT[176]), pointed to by HWRPB[160].

**D.2 Architecture Mask and Implementation Value Assignments**

The following bits are defined for the AMASK instruction.

**Table D–2: AMASK Bit Assignments**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Support for the byte/word extension (BWX)</td>
</tr>
<tr>
<td></td>
<td>The instructions that comprise the BWX extension are LDBU, LDWU, SEXTB, SEXTW, STB, and STW.</td>
</tr>
<tr>
<td>1</td>
<td>Support for the count extension (CIX)</td>
</tr>
<tr>
<td></td>
<td>The instructions that comprise the CIX extension are CTLZ, CTPOP, CTTZ, FTOIS, FTOIT, ITOFF, ITOFS, ITOFT, SQRTF, SQRTG, SQRTS, and SQRTT.</td>
</tr>
<tr>
<td>8</td>
<td>Support for the multimedia extension (MAX)</td>
</tr>
<tr>
<td></td>
<td>The instructions that comprise the MAX extension are MAXSB8, MAXSW4, MAXUB8, MAXUW4, MINSB8, MINSW4, MINUB8, MINUW4, PERR, PKLB, PKWB, UNPKBL, and UNPKBW.</td>
</tr>
</tbody>
</table>

The following values are defined for the IMPLVER instruction.

**Table D–3: IMPLVER Value Assignments**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21064 (EV4)</td>
</tr>
<tr>
<td></td>
<td>21064A (EV45)</td>
</tr>
<tr>
<td></td>
<td>21066A/21068A (LCA45)</td>
</tr>
<tr>
<td>1</td>
<td>21164 (EV5)</td>
</tr>
<tr>
<td></td>
<td>21164A (EV56)</td>
</tr>
<tr>
<td></td>
<td>21164PC (PCA56)</td>
</tr>
<tr>
<td>2</td>
<td>21264 (EV6)</td>
</tr>
</tbody>
</table>
Index

21064/21064A processors, IMPLVER values for, D–1
21066/21066A processors, IMPLVER values for, D–1
21164/21164A processors, IMPLVER values for, D–1
21164PC processor, IMPLVER values for, D–1
21264 processor, IMPLVER values for, D–1

A

Aborts, forcing, 6–6
ACCESS(x,y) operator, 3–6
ADDF instruction, 4–104
ADDG instruction, 4–104
Add instructions
See also Floating-point operate
add longword, 4–26
add quadword, 4–28
add scaled longword, 4–27
add scaled quadword, 4–29
ADDL instruction, 4–26
ADDQ instruction, 4–28
Address space match (ASM)
virtual cache coherency, 5–4
Address space number (ASN) register
virtual cache coherency, 5–4
ADDs instruction, 4–105
ADDT instruction, 4–105
AFTER, defined for memory access, 5–12
Aligned byte/word memory accesses, A–11
ALIGNED data objects, 1–9
Alignment
atomic byte, 5–2
atomic longword, 5–2
atomic quadword, 5–2
data considerations, A–6
double-width data paths, A–1
D_floating, 2–6
F_floating, 2–4
G_floating, 2–5
instruction, A–2
longword, 2–2
longword integer, 2–12
memory accesses, A–11
quadword, 2–3
quadword integer, 2–13
S_floating, 2–9
T_floating, 2–10
Alignment (cont'd)
X_floating, 2–11
Alpha architecture
See also Conventions
addressing, 2–1
overview, 1–1
porting operating systems to, 1–1
programming implications, 5–1
registers, 3–1
security, 1–7
Alpha privileged architecture library
See PALcode
AMASK (Architecture mask) instruction, 4–132
bit assignments for, D–1
IMPLVER compared to, 4–140
AND instruction, 4–43
AND operator, 3–6
Architecture extensions, D–1
AMASK with, 4–132
Arithmetic instructions, 4–25
See also specific arithmetic instructions
Arithmetic left shift instruction, 4–42
Arithmetic traps
disabling, 4–73
division by zero, 4–72, 4–75
division by zero, disabling, 4–75
division by zero, enabling, B–5
division by zero, status of, B–5
dynamic rounding mode, 4–75
enabling, B–4
inexact result, 4–72, 4–75
inexact result, disabling, 4–74
inexact result, enabling, B–5
inexact result, status of, B–4
integer overflow, 4–73, 4–75
integer overflow, disabling, B–4
integer overflow, enabling, B–4
invalid operation, 4–71, 4–75
invalid operation, disabling, 4–75
invalid operation, enabling, B–5
invalid operation, status of, B–5
overflow, 4–72, 4–75
overflow, disabling, 4–75
overflow, enabling, B–5
overflow, status of, B–5
programming implications for, 5–29
Arithmetic traps (cont’d)
  TRAPB instruction with, 4–143
  underflow, 4–72
  underflow, disabling, 4–74
  underflow, enabling, B–5
  underflow, status of, B–5
  underflow to zero, disabling, 4–74
ARITH_RIGHT_SHIFT(x,y) operator, 3–6
ASCII character set, C–18
Atomic access, 5–3
Atomic operations
  accessing longword datum, 5–2
  accessing quadword datum, 5–2
  updating shared data structures, 5–7
  using load locked and store conditional, 5–7
Atomic sequences, A–18

B
BEFORE, defined for memory access, 5–12
BEQ instruction, 4–20
BGE instruction, 4–20
BGT instruction, 4–20
BIC instruction, 4–43
Big-endian addressing, 2–14
  byte operation examples, 4–55
  byte swapping for, A–12
  extract byte with, 4–52
  insert byte with, 4–56
  load F_floating with, 4–85
  load long/quad locked with, 4–9
  load long/quad with, 4–6
  load S_floating with, 4–87
  mask byte with, 4–58
  store byte/word with, 4–15
  store F_floating with, 4–89
  store long/quad conditional with, 4–12
  store long/quad with, 4–15
  store S_floating with, 4–91
BIS instruction, 4–43
BLBCE instruction, 4–20
BLBS instruction, 4–20
BLE instruction, 4–20
BLT instruction, 4–20
BNE instruction, 4–20
Boolean instructions, 4–42
  logical functions, 4–43
  Boolean stylized code forms, A–15
  bpt (PALcode) instruction, 10–1, 11–1
    required recognition of, 6–4
  BPT (PALcode) instruction, 9–1
    required recognition of, 6–4
Branch instructions, 4–19
  See also Control instructions
  backward conditional, 4–20
  conditional branch, 4–20
  displacement, 4–21
  floating-point, summarized, 4–93
  format of, 3–11
  forward conditional, 4–20
  opcodes and format summarized, C–1
  unconditional branch, 4–22
  with trap shadow, 4–71
Branch prediction model, 4–18
Branch prediction stack, with BSR instruction, 4–22
BR instruction, 4–22
BSR instruction, 4–22
bugchk (PALcode) instruction, 10–1
  required recognition of, 6–4
BUGCHK (PALcode) instruction, 9–1
  required recognition of, 6–4
BWX extension
  See also the Preface
  AMASK instruction with, D–1
Byte data type, 2–1
  atomic access of, 5–2
Byte manipulation, 1–2
Byte manipulation instructions, 4–48
  LDBU, 4–6
Byte swapping, A–12
BYTE_ZAP(x,y) operator, 3–6

C
Cache coherency
  barrier instructions for, 5–25
  defined, 5–2
  in multiprocessor environment, 5–6
Caches
  design considerations, A–1
  I-stream considerations, A–5
  MB and IMB instructions with, 5–25
  requirements for, 5–5
  translation buffer conflicts, A–8
  with powerfail/recovery, 5–5
callkd (PALcode) instruction, 11–1
callsys (PALcode) instruction, 10–1
CALL_PAL (call privileged architecture library) instruction, 4–134
Canonical form, 4–63
CASE operator, 3–7
Causal loops, 5–15
cflush (PALcode) instruction, 10–2
CFLUSH (PALcode) instruction, 9–7
  ECB compared with, 4–136
Changed datum, 5–6
CHME (PALcode) instruction, 9–1
CHMK (PALcode) instruction, 9–1
CHMS (PALcode) instruction, 9–2
CHMU (PALcode) instruction, 9–2
CIX extension
   See also the Preface
   AMASK instruction with, D–1
Clear a register, A–13
clrfrn (PALcode) instruction, 10–1
CLRFEN (PALcode) instruction, 9–2
CMOVEQ instruction, 4–44
CMOVGE instruction, 4–44
CMOVGT instruction, 4–44
CMOVLBC instruction, 4–44
CMOVLBS instruction, 4–44
CMOVLE instruction, 4–44
CMOVLT instruction, 4–44
CMOVEQ instruction, 4–44
CMOVLE instruction, 4–44
CMOVLT instruction, 4–44
CMOVNE instruction, 4–44
CMOVLT instruction, 4–44
CMPBGE instruction, 4–50
CMPEQ instruction, 4–30
CMPGEQ instruction, 4–106
CMPPLE instruction, 4–106
CMPPGT instruction, 4–106
CMPPLE instruction, 4–30
CMPPLT instruction, 4–30
CMPTEQ instruction, 4–107
CMPTLE instruction, 4–107
CMPTLT instruction, 4–107
CMPTUN instruction, 4–107
CMPPLE instruction, 4–31
CMPPULT instruction, 4–31
Code forms, stylized, A–13
   Boolean, A–15
   load literal, A–14
   negate, A–14
   NOP, A–13
   NOT, A–15
   register, clear, A–13
   register-to-register move, A–14
Code scheduling
   IMPLVER instruction with, 4–140
Code sequences, A–11
Coherency
   cache, 5–2
   memory, 5–1
Compare instructions
   See also Floating-point operate
   compare byte, 4–50
   compare integer signed, 4–30
   compare integer unsigned, 4–31
Conditional move instructions, 4–44
   See also Floating-point operate
   with trap shadow, 4–70
Console overview, 7–1
Context switching
   multiprocessor considerations, 5–24
Control instructions, 4–18
Conventions
   code examples, 1–9
   extents, 1–8
   figures, 1–9
   instruction format, 3–9
   notation, 3–9
   numbering, 1–7
   ranges, 1–8
/C opcode qualifier
   IEEE floating-point, 4–67
   VAX floating-point, 4–67
Count instructions
   Count leading zero, 4–32
   Count population, 4–33
   Count trailing zero, 4–34
CPYSE instruction, 4–99
CPYS instruction, 4–99
CPYSN instruction, 4–99
cserve (PALcode) instruction, 10–2
   required recognition of, 6–4
CSERVE (PALcode) instruction, 9–7
   required recognition of, 6–4
csir (PALcode) instruction, 11–2
CTLZ instruction, 4–32
CTPOP instruction, 4–33
CTTZ instruction, 4–34
CVTQG instruction, 4–109
CVTQG instruction, 4–110
CVTQG instruction, 4–100
CVTQG instruction, 4–115
FP_C quadword with, B–4
CVTST instruction, 4–116
CVTTQ instruction, 4–113
FP_C quadword with, B–4
CVTTS instruction, 4–115

D

Data alignment, A–6
Data caches
   ECB instruction with, 4–135
   WH64 instruction with, 4–144
Data fetches (memory), 5–11

Index–3
Data format, overview, 1-3
Data sharing (multiprocessor), A–7
synchronization requirement, 5–6
Data stream considerations, A–6
Data structures, shared, 5–6
Data types
  byte, 2–1
  IEEE floating-point, 2–7
  longword, 2–2
  longword integer, 2–12
  quadword, 2–2
  quadword integer, 2–13
  unsupported in hardware, 2–13
  VAX floating-point, 2–3
  word, 2–2
Denormal, 4–64
Depends order (DP), 5–15
di (PALcode) instruction, 11–2
Digital UNIX PALcode, instruction summary, C–18
Digital UNIX privileged PALcode instructions, 10–2
Digital UNIX unprivileged PALcode instructions, 10–1
Dirty zero, 4–64
DIVF instruction, 4–117
DIVG instruction, 4–117
Division
  integer, A–12
  performance impact of, A–12
Division by zero enable (DZEE)
  FP.C quadword bit, B–5
Division by zero status (DZES)
  FP.C quadword bit, B–5
DIV operator, 3–7
DIVS instruction, 4–119
DIVT instruction, 4–119
/D opcode qualifier
  FPCR (floating-point control register), 4–73
  IEEE floating-point, 4–67
DP
  See Depends order
draina (PALcode) instruction, 10–2, 11–2
  required, 6–6
DRAINa (PALcode) instruction, 9–7
  required, 6–6
dtbis (PALcode) instruction, 11–2
Dual-issue instruction considerations, A–2
DYN bit
  See Arithmetic traps, dynamic rounding mode
DZE bit
  See Arithmetic traps, division by zero
DZED bit
  See Trap disable bits, division by zero
D_floating data type, 2–6
  alignment of, 2–6
  mapping, 2–6
  restricted, 2–6

E
ECB (Evict data cache block) instruction, 4–135
CLUSH (PALcode) instruction with, 4–136
ei (PALcode) instruction, 11–2
EQV instruction, 4–43
EXCB (exception barrier) instruction, 4–137
  with FPCR, 4–78
Exception handlers, B–2
  TRAPB instruction with, 4–143
Exceptions
  F31 with, 3–2
  R31 with, 3–1
EXTBL instruction, 4–52
Extensions to architecture, D–1
EXTLH instruction, 4–52
EXTLL instruction, 4–52
EXTQH instruction, 4–53
EXTQL instruction, 4–52
Extract byte instructions, 4–52
EXTWH instruction, 4–52
EXTWL instruction, 4–52

F
FBEQ instruction, 4–94
FBGE instruction, 4–94
FBGT instruction, 4–94
FBLE instruction, 4–94
FBLT instruction, 4–94
FBNE instruction, 4–94
FCMOVEQ instruction, 4–101
FCMOVGE instruction, 4–101
FCMOVGT instruction, 4–101
FCMOVL instruction, 4–101
FCMOVLT instruction, 4–101
FCMOVNE instruction, 4–101
FETCH (prefetch data) instruction, 4–138
  performance optimization, A–10
FETCH_M (prefetch data, modify intent) instruction, 4–138
  performance optimization, A–10
Finite number, Alpha, contrasted with VAX, 4–65
Floating-point branch instructions, 4–93
Floating-point control register (FPCR), 4–73
     accessing, 4–76
     at processor initialization, 4–77
Floating-point control register (FPCR) (cont’d)
  bit descriptions, 4–74
  instructions to read/write, 4–103
  operate instructions that use, 4–96
  saving and restoring, 4–77
  trap disable bits in, 4–73
Floating-point convert instructions, 3–13
  F_a field requirements, 3–13
Floating-point division, performance impact of, A–12
Floating-point format, number representation (encodings), 4–65
Floating-point instructions
  branch, 4–93
  faults, 4–63
  function field format, 4–79
  introduced, 4–63
  memory format, 4–84
  opcodes and format summarized, C–1
  operate, 4–96
  rounding modes, 4–66
  terminology, 4–64
  trapping modes, 4–68
  traps, 4–63
Floating-point load instructions, 4–84
  load F_floating, 4–85
  load G_floating, 4–86
  load S_floating, 4–87
  load T_floating, 4–88
  with non-finite values, 4–84
Floating-point operate instructions, 4–96
  add (IEEE), 4–105
  add (VAX), 4–104
  compare (IEEE), 4–107
  compare (VAX), 4–106
  conditional move, 4–101
  convert IEEE floating to integer, 4–113
  convert integer to IEEE floating, 4–114
  convert integer to integer, 4–100
  convert integer to VAX floating, 4–110
  convert S_floating to T_floating, 4–115
  convert T_floating to S_floating, 4–116
  convert VAX floating to integer, 4–109
  convert VAX floating to VAX floating, 4–111
  copy sign, 4–99
  divide (IEEE), 4–119
  divide (VAX), 4–117
  format of, 3–12
  from integer moves, 4–122
  move from/to FPCR, 4–103
  multiply (IEEE), 4–125
  multiply (VAX), 4–124
  subtract (IEEE), 4–130
  subtract (VAX), 4–128
  to integer moves, 4–120
Floating-point operate instructions (cont’d)
  unused function codes with, 3–12
Floating-point registers, 3–2
Floating-point rounding modes, 4–66
Floating-point single-precision operations, 4–63
Floating-point store instructions, 4–84
  store F_floating, 4–89
  store G_floating, 4–90
  store S_floating, 4–91
  store T_floating, 4–92
  with non-finite values, 4–84
Floating-point support
  floating-point control (FP_C) quadword, B–4
  FPCR (floating-point control register), 4–73
  IEEE, 2–7
  IEEE standard 754-1985, 4–82
  instruction overview, 4–63
  longword integer, 2–12
  operate instructions, 4–96
  optional, 4–2
  quadword integer, 2–13
  rounding modes, 4–66
  single-precision operations, 4–63
  trap modes, 4–68
  VAX, 2–3
Floating-point to integer move, 4–120
Floating-point to integer move instructions, 3–13
Floating-point trapping modes, 4–68
  See also Arithmetic traps
  imprecision from pipelining, 4–69
FNOP code form, A–13
FPCR
  See Floating-point control register
FP_C quadword, B–4
FTOIS instruction, 4–120
FTOIT instruction, 4–120
Function codes
  See also Opcodes
  IEEE floating-point, C–7
  independent floating-point, C–9
  in numerical order, C–13
  VAX floating-point, C–9
F_floating data type, 2–3
  alignment of, 2–4
  compared to IEEE S_floating, 2–8
  MAX/MIN, 4–66
  operations, 4–63

G

Index–5
gentrap (PALcode) instruction, 10–1, 11–1
required recognition of, 6–4
GENTRAP (PALcode) instruction, 9–2
required recognition of, 6–4
G_floating data type, 2–5
alignment of, 2–5
mapping, 2–5
MAX/MIN, 4–66

H

halt (PALcode) instruction, 10–2, 11–2
required, 6–8
HALT (PALcode) instruction, 9–7
required, 6–8

I

I/O devices, DMA
MB and WMB with, 5–23
reliably communicating with processor, 5–27
shared memory locations with, 5–11
I/O interface overview, 8–1
IEEE compliance, B–2
IEEE convert-to-integer trap mode,
instruction notation for, 4–69
IEEE floating-point
See also Floating-point instructions
exception handlers, B–2
floating-point control (FP_C) quadword,
format, 2–7
FPCR (floating-point control register), 4–73
function field format, 4–80
hardware support, B–1
NaN, 2–7
options, B–1
standard, mapping to, B–5
standard charts, B–12
S_floating, 2–7
trap handling, B–6
trap modes, 4–69
T_floating, 2–9
X_floating, 2–10
IEEE floating-point compliance, 4–78
IEEE floating-point control word, B–4
IEEE floating-point instructions
add instructions, 4–105
compare instructions, 4–107
convert from integer instructions, 4–114
convert S_floating to T_floating, 4–115
convert to integer instructions, 4–113
convert T_floating to S_floating, 4–116
divide instructions, 4–119
from integer moves, 4–122
IEEE floating-point instructions (cont’d)
function codes for, C–7
multiply instructions, 4–125
operate instructions, 4–96
square root instructions, 4–127
subtract instructions, 4–130
to register moves, 4–120
IEEE rounding modes, 4–66
IEEE standard
conformance to, B–1
mapping to, B–5
support for, 4–78
IEEE trap modes, required instruction
notation, 4–69
IGN (ignore), 1–9
imb (PALcode) instruction, 10–1, 11–1
required, 6–9
IMB (PALcode) instruction, 5–23, 9–2
required, 6–9
virtual I-cache coherency, 5–5
IMP (implementation dependent), 1–9
IMPLVER (Implementation version)
instruction, 4–140
value assignments for, D–1
Independent floating-point function codes,
C–9
INE bit
See Arithmetic traps, inexact result
INED bit
See Trap disable bits, inexact result trap
Inexact result enable (INEE)
FP_C quadword bit, B–5
Inexact result status (INES)
FP_C quadword bit, B–4
Infinity, 4–64, 4–65
corversion to integer, 4–82
initpal (PALcode) instruction, 11–3
INSBL instruction, 4–56
Insert byte instructions, 4–56
INSLH instruction, 4–56
INSLL instruction, 4–56
INSQHIL (PALcode) instruction, 9–2
INSQHILR (PALcode) instruction, 9–2
INSQH instruction, 4–56
INSQHIQ (PALcode) instruction, 9–3
INSQHIQR (PALcode) instruction, 9–3
INSQHIQ (PALcode) instruction, 9–3
INSQH instruction, 4–56
INSQHIQ (PALcode) instruction, 9–3
INSQHIQR (PALcode) instruction, 9–3
INSQHILR (PALcode) instruction, 9–2
Instruction encodings
  common architecture, C-1
  numerical order, C-13
  opcodes and format summarized, C-1
Instruction fetches (memory), 5-11
Instruction formats
  branch, 3-11
  conventions, 3-9
  floating-point convert, 3-13
  floating-point operate, 3-12
  floating-point to integer move, 3-13
  memory, 3-9
  memory jump, 3-10
  operands, 3-9
  operand values, 3-9
  operate, 3-11
  operators, 3-6
  overview, 1-4
  PALcode, 3-13
  registers, 3-1
Instructions, overview, 1-5
Instruction set
  See also Floating-point instructions;
    PALcode instructions
  access type field, 3-5
  Boolean, 4-42
  branch, 4-19
  byte manipulate, 4-48
  conditional move (integer), 4-44
  data type field, 3-5
  floating-point subsetting, 4-2
  integer arithmetic, 4-25
  introduced, 1-6
  jump, 4-19
  load memory integer, 4-4
  miscellaneous, 4-131
  multimedia, 4-151
  name field, 3-4
  opcode qualifiers, 4-3
  operand notation, 3-4
  overview, 4-1
  shift, arithmetic, 4-47
  shift, logical, 4-46
  software emulation rules, 4-3
  store memory integer, 4-4
  VAX compatibility, 4-149
Instruction stream
  See I-stream
  INSWH instruction, 4-56
  INSWL instruction, 4-56
Integer arithmetic instructions
  See Arithmetic instructions

Integer division, A-12
Integer registers
  defined, 3-1
  R31 restrictions, 3-1
Invalid operation enable (INVE)
  FP_C quadword bit, B-5
Invalid operation status (INVS)
  FP_C quadword bit, B-5
INV bit
  See also Arithmetic traps, invalid operation
INVD bit
  See Trap disable bits, invalid operation
/I opcode qualifier, IEEE floating-point, 4-69
IOV bit
  See Arithmetic traps, integer overflow
I-stream
  coherency of, 6-9
  design considerations, A-2
  modifying physical, 5-5
  modifying virtual, 5-5
  PALcode with, 6-2
  with caches, 5-5
ITOFF instruction, 4-122
ITOFS instruction, 4-122
ITOFT instruction, 4-122

J
  JMP instruction, 4-23
  JSR instruction, 4-23
  JSR_COROUTINE instruction, 4-23
  Jump instructions, 4-19, 4-23
    See also Control instructions
    branch prediction logic, 4-24
    coroutine linkage, 4-24
    return from subroutine, 4-23
    unconditional long jump, 4-24

K
  kbpt (PALcode) instruction, 11-2

L
  LDAH instruction, 4-5
  LDA instruction, 4-5
  LDBU instruction, 4-6
  LDF instruction, 4-85
  LDG instruction, 4-86
  LDL instruction, 4-6
  LDL_L instruction, 4-9
    restrictions, 4-10
    with processor lock register/flag, 4-10
    with STx_C instruction, 4-9
LDQ instruction, 4–6
LDQP (PALcode) instruction, 9–7
LDQ_L instruction, 4–9
restrictions, 4–10
with processor lock register/flag, 4–10
with STx C instruction, 4–10
LDQ_U instruction, 4–8
LDS instruction, 4–87
with FPCR, 4–78
LDT instruction, 4–88
LDWU instruction, 4–6
LEFT SHIFT(x,y) operator, 3–7
Ig operator, 3–7
Literals, operand notation, 3–4
Litmus tests, shared data veracity, 5–17
Load instructions
See also Floating-point load instructions
emulation of, 4–3
FETCH instruction, 4–138
load address, 4–5
load address high, 4–5
load quadword, 4–6
load quadword locked, 4–10
load sign-extended longword, 4–6
load sign-extended longword locked, 4–9
load unaligned quadword, 4–8
Load zero-extended byte, 4–6
Load zero-extended word, 4–6
multiprocessor environment, 5–6
serialization, 4–141
Load literal, A–14
Load-locked, defined, 5–16
Load memory instructions, with trap shadow, 4–71
Load memory integer instructions, 4–4
LOAD_LOCKED operator, 3–7
Location, 5–11
Location access constraints, 5–14
Lock flag, per-processor
defined, 3–2
when cleared, 4–10
with load locked instructions, 4–10
with store conditional instructions, 4–12
Lock registers, per-processor
defined, 3–2
with load locked instructions, 4–10
with store conditional instructions, 4–12
Lock variables, with WMB instruction, 4–147
Logical instructions
See Boolean instructions
Longword data type, 2–2
alignment of, 2–12
atomic access of, 5–2
integer floating-point format, 2–12
LSB (least significant bit), defined for floating-point, 4–65

M

MAP_F function, 2–4
MAP_S function, 2–8
MAP_x operator, 3–7
Mask byte instructions, 4–58
MAX, defined for floating-point, 4–66
MAX extension
See also the Preface
AMASK instruction with, D–1
MAXS(x,y) operator, 3–7
MAXSB8 instruction, 4–152
MAXSW4 instruction, 4–152
MAXU(x,y) operator, 3–7
MAXUB8 instruction, 4–152
MAXUW4 instruction, 4–152
MB (Memory barrier) instruction, 4–141
See also IMB, WMB
compared with WMB, 4–147
multiprocessors only, 4–141
using, 5–23
with DMA I/O, 5–23
with LDx L/STx C, 4–13
with multiprocessor D-stream, 5–22
with shared data structures, 5–9
MBZ (must be zero), 1–9
Memory access
aligned byte/word, A–11
coherency of, 5–1
granularity of, 5–2
width of, 5–3
with WMB instruction, 4–146
Memory alignment, requirement for, 5–2
Memory barrier instructions
See MB, IMB (PALcode), and WMB instructions
Memory barriers, 5–22
Memory format instructions
opcodes and format summarized, C–1
Memory instruction format, 3–9
with function code, 3–10
Memory jump instruction format, 3–10
Memory-like behavior, 5–3
Memory management
support in PALcode, 6–2
Memory prefetch registers, A–10
defined, 3–3
MFPR (PALcode) instruction, 9–7
MF_FPCR instruction, 4–103
MIN, defined for floating-point, 4–66
**MINS(x,y) operator**, 3–7
**MINSB8 instruction**, 4–152
**MINSW4 instruction**, 4–152
**MINU(x,y) operator**, 3–7
**MINUB8 instruction**, 4–152
**MINUW4 instruction**, 4–152
**Miscellaneous instructions**, 4–131
/M opcode qualifier, IEEE floating-point, 4–67
**Move, register-to-register**, A–14

**Move instructions (conditional)**

- See Conditional move instructions
- **MSKBL instruction**, 4–58
- **MSKLH instruction**, 4–58
- **MSKLL instruction**, 4–58
- **MSKQL instruction**, 4–58
- **MSKWH instruction**, 4–58
- **MSKWL instruction**, 4–58
- **MTPR (PALcode) instruction**, 9–7
- **MT_FPCR instruction**, 4–103
  - synchronization requirement, 4–76
- **MULF instruction**, 4–124
- **MULG instruction**, 4–124
- **MULL instruction**, 4–35
  - with MULQ, 4–35
- **MULQ instruction**, 4–36
  - with MULL, 4–35
  - with UMULH, 4–36
- **MULS instruction**, 4–125
- **Multimedia instructions**, 4–151
- **MULT instruction**, 4–125
- **Multiple instruction issue**, A–2

**Multiply instructions**

- See also Floating-point operate
- multiply longword, 4–35
- multiply quadword, 4–36
- multiply unsigned quadword high, 4–37

**Multiprocessor environment**

- See also Data sharing
- cache coherency in, 5–6
- context switching, 5–24
- l-stream reliability, 5–23
- MB and WMB with, 5–23
- no implied barriers, 5–22
- read/write ordering, 5–10
- serialization requirements in, 4–141
- shared data, 5–6, A–7

**N**

**NaN (Not-a-Number)**

- conversion to integer, 4–82
- copying, generating, propagating, 4–83
  - defined, 2–7
  - quiet, 4–65
  - signaling, 4–65

**NATURALLY ALIGNED data objects**, 1–9
**Negate stylized code form**, A–14
**Non-finite number**, 4–65
**Nonmemory-like behavior**, 5–3
**NOP, universal (UNOP)**, A–13
**NOT instruction, ORNOT with zero**, 4–43
**NOT operator**, 3–7
**NOT stylized code form**, A–15

**O**

**Opcode qualifiers**

- See also specific qualifiers
- default values, 4–3
- notation, 4–3

**Opcodes**

- See also Function codes
- common architecture, C–1
- Digital UNIX PALcode, C–18
  - in numerical order, C–13
  - notation used in summary, C–1
- OpenVMS Alpha PALcode, C–17
  - PALcode in numerical order, C–18
  - reserved, C–18
  - summary, C–11
  - unused function codes for, C–18
- Windows NT Alpha PALcode, C–18

**OpenVMS Alpha PALcode, instruction summary**, C–17
**OpenVMS Alpha privileged PALcode instructions**, 9–7
**OpenVMS Alpha unprivileged PALcode instructions**, 9–1
**Operand expressions**, 3–4
**Operand notation defined**, 3–3
**Operand values**, 3–4
**Operate instruction format**, 3–11
  - floating-point, 3–12
  - floating-point convert, 3–13
  - unused function codes with, 3–11
**Operate instructions**

- opcodes and format summarized, C–1
- Operate instructions, convert with integer overflow, 4–73
**Operators, instruction format**, 3–6
**Optimization**

- See Performance optimizations
**ORNOT instruction**, 4–43
**OR operator**, 3–7
**Overflow enable (OVFE)**
  - FP_C quadword bit, B–5
**Overflow status (OVFS)**
  - FP_C quadword bit, B–5

Index–9
Overlap
  with location access constraints, 5–14
  with processor issue constraints, 5–13
  with visibility, 5–14
OVF bit
  See Arithmetic traps, overflow
OVFD bit
  See Trap disable bits, overflow disable

P
Pack to bytes instructions, 4–155
PALcode
  barriers with, 5–22
  CALL_PAL instruction, 4–134
  compared to hardware instructions, 6–1
  defined for Digital UNIX, 10–1
  Digital-defined for OpenVMS Alpha, 9–1
  Digital-defined for Windows NT Alpha, 11–1
  implementation-specific, 6–2
  instead of microcode, 6–1
  instruction format, 3–13
  overview, 6–1
  privileged Digital UNIX, 10–2
  privileged OpenVMS Alpha, 9–7
  privileged Windows NT Alpha, 11–2
  recognized instructions, 6–4
  replacing, 6–3
  required function support, 6–2
  required instructions, 6–5
  running environment, 6–2
  special functions, 6–2
  unprivileged Digital UNIX, 10–1
  unprivileged OpenVMS Alpha, 9–1
  unprivileged Windows NT Alpha, 11–1
PALcode instructions
  opcodes and format summarized, C–1
  required, opcodes for, C–18
  reserved, opcodes for, C–18
PALcode instructions, required privileged, 6–6
PALcode instructions, required unprivileged, 6–9
PALcode Opcodes in numerical order, C–18
PCC_CNT, 3–2, 4–142
PCC_OFF, 3–2, 4–142
Performance optimizations
  branch prediction, A–3
  code sequences, A–11
  data stream, A–6
  for frequently executed code, A–1
  for I-streams, A–2
  instruction alignment, A–2
  instruction scheduling, A–5
Performance optimizations (cont’d)
  I-stream density, A–5
  multiple instruction issue, A–2
  shared data, A–7
Performance tuning
  IMPLVER instruction with, 4–140
  PERR (Pixel error) instruction, 4–154
Physical address space
  described, 5–1
  PHYSICAL_ADDRESS operator, 3–8
Pipelined implementations, using EXCB
  instruction with, 4–137
Pixel error instruction, 4–154
PKLB (Pack longwords to bytes) instruction, 4–155
PKWB (Pack words to bytes) instruction, 4–155
Prefetch data (FETCH instruction), 4–138
Prefetch data registers, A–10
PRIORITY_ENCODE operator, 3–8
Privileged Architecture Library
  See PALcode
PROBE (PALcode) instruction, 9–4
Processor communication, 5–15
Processor cycle counter (PCC) register, 3–2
  RPCC instruction with, 4–142
Processor issue constraints, 5–13
Processor issue sequence, 5–12
Processor type assignments, D–1
Program counter (PC) register, 3–1
  with EXCB instruction, 4–137
Pseudo-ops, A–15
Q
Quadword data type, 2–2
  alignment of, 2–3, 2–13
  atomic access of, 5–2
  integer floating-point format, 2–13
  T_floating with, 2–13
R
R31
  restrictions, 3–1
RAZ (read as zero), 1–9
RC (read and clear) instruction, 4–150
rdirql (PALcode) instruction, 11–3
rdksp (PALcode) instruction, 11–3
rdmces (PALcode) instruction, 10–2, 11–3
rdpcr (PALcode) instruction, 11–3
rdps (PALcode) instruction, 10–2
rdpsr (PALcode) instruction, 11–3
rdteb (PALcode) instruction, 11–2
Index–11

rdthread (PALcode) instruction, 11–3
rdunique (PALcode) instruction, 10–1
required recognition of, 6–5
RDUNIQUE (PALcode) instruction
required recognition of, 6–5
rdusp (PALcode) instruction, 10–2
rdval (PALcode) instruction, 10–2
RD_PS (PALcode) instruction, 9–4
Read/write, sequential, A–9
Read/write ordering (multiprocessor), 5–10
determining requirements, 5–10
hardware implications for, 5–28
memory location defined, 5–11
READ_UNQ (PALcode) instruction, 9–4
reboot (PALcode) instruction, 11–3
Regions in physical address space, 5–1
Registers, 3–1
See also specific registers
floating-point, 3–2
integer, 3–1
lock, 3–2
memory prefetch, 3–3
optional, 3–3
processor cycle counter, 3–2
program counter (PC), 3–1
value when unused, 3–9
VAX compatibility, 3–3
Register-to-register move, A–14
REI (PALcode) instruction, 9–4
Relational Operators, 3–8
REMQL (PALcode) instruction, 9–4
REMQLILR (PALcode) instruction, 9–5
REMQLILQ (PALcode) instruction, 9–5
REMQLILQR (PALcode) instruction, 9–5
REMQLTIL (PALcode) instruction, 9–5
REMQLTILI (PALcode) instruction, 9–5
REMQLTIQ (PALcode) instruction, 9–5
REMQLTILR (PALcode) instruction, 9–5
REMQLTIQR (PALcode) instruction, 9–6
REMQUEL (PALcode) instruction, 9–6
REMQUELQ (PALcode) instruction, 9–6
Remainder:
result, 4–65
Reserved instructions, opcodes for, C–18
Reserved operand, 4–65
restart (PALcode) instruction, 11–3
Result latency, A–5
RET instruction, 4–23
rets (PALcode) instruction, 10–2, 11–4
rfe (PALcode) instruction, 11–4
RIGHT_SHIFT(x,y) operator, 3–8
Rounding modes
See Floating-point rounding modes
RPCC (Read processor cycle counter)
instruction, 4–142
RS (read and set) instruction, 4–150
RSCC (PALcode) instruction, 9–6
rti (PALcode) instruction, 10–2
S
S4ADDL instruction, 4–27
S4ADDQ instruction, 4–29
S4SUBL instruction, 4–39
S4SUBQ instruction, 4–41
S8ADDL instruction, 4–27
S8ADDQ instruction, 4–29
S8SUBL instruction, 4–39
S8SUBQ instruction, 4–41
SBZ (should be zero), 1–9
Security holes, 1–7
with UNPREDICTABLE results, 1–8
Sequential read/write, A–9
Serialization, MB instruction with, 4–141
SEXT(x) operator, 3–8
SEXTB instruction, 4–61
SEXTW instruction, 4–61
Shared data (multiprocessor), A–7
changed vs. updated datum, 5–6
Shared data structures
atomic update, 5–7
ordering considerations, 5–9
using memory barrier (MB) instruction,
5–9
Shared memory
accessing, 5–11
access sequence, 5–11
defined, 5–11
issue sequence, 5–11
Shift arithmetic instructions, 4–47
Shift logical instructions, 4–46
Sign extend instructions, 4–61
Single-precision floating-point, 4–63
SLL instruction, 4–46
Software considerations, A–1
See also Performance optimizations
/S opcode qualifier
IEEE floating-point, 4–69
VAX floating-point, 4–68
SQRFT instruction, 4–126
SQRTG instruction, 4–126
SQRTS instruction, 4–127
SQRTT instruction, 4–127
Square root instructions
IEEE, 4–127
VAX, 4–126

Index–11
SRA instruction, 4–47
SRL instruction, 4–46
ssir (PALcode) instruction, 11–4
STB instruction, 4–15
STF instruction, 4–89
STG instruction, 4–90
STL instruction, 4–15
STLC instruction, 4–12
when guaranteed ordering with LDL_L, 4–13
with LDx_L instruction, 4–12
with processor lock register/flag, 4–12
Storage, defined, 5–14
Store-conditional, defined, 5–16
Store instructions
See also Floating-point store instructions
emulation of, 4–3
FETCH instruction, 4–138
multiprocessor environment, 5–6
serialization, 4–141
Store byte, 4–15
store longword, 4–15
store longword conditional, 4–12
store quadword, 4–15
store quadword conditional, 4–12
store unaligned quadword, 4–17
Store word, 4–15
Store memory instructions, with trap shadow, 4–71
Store memory integer instructions, 4–4
STORE_CONDITIONAL operator, 3–8
STQ instruction, 4–15
STQP (PALcode) instruction, 9–7
STQC instruction, 4–12
when guaranteed ordering with LDQ_L, 4–13
with LDx_L instruction, 4–12
with processor lock register/flag, 4–12
STQU instruction, 4–17
STSS instruction, 4–91
with FPCR, 4–78
STTW instruction, 4–15
SUBF instruction, 4–128
SUBG instruction, 4–128
SUBL instruction, 4–38
SUBQ instruction, 4–40
SUBS instruction, 4–130
SUBT instruction, 4–130
Subtract instructions
See also Floating-point operate
subtract longword, 4–38
subtract quadword, 4–40
subtract scaled longword, 4–39
subtract scaled quadword, 4–41
SUM bit
See Summary bit
Summary bit, in FPCR, 4–74
SWASTEN (PALcode) instruction, 9–6
swpctx (PALcode) instruction, 10–3, 11–4
SWPCTX (PALcode) instruction, 9–8
swpipl (PALcode) instruction, 10–3
swpirq (PALcode) instruction, 11–4
swpksp (PALcode) instruction, 11–4
swppal (PALcode) instruction, 10–3, 11–4
required recognition of, 6–5
SWPPAL (PALcode) instruction, 9–8
required recognition of, 6–5
swpprocess (PALcode) instruction, 11–4
S_floating data type
alignment of, 2–9
compared to F_floating, 2–8
exceptions, 2–8
format, 2–8
mapping, 2–8
MAX/MIN, 4–66
NaN with T_floating convert, 4–82
operations, 4–63
T

tbi (PALcode) instruction, 10–3
tbia (PALcode) instruction, 11–4
tbis (PALcode) instruction, 11–5
tbisasn (PALcode) instruction, 11–5
TEST(x,cond) operator, 3–8
Timing considerations, atomic sequences, A–18
TRAPB (trap barrier) instruction, A–15
described, 4–143
with FPCR, 4–78
with trap shadow, 4–71
Trap disable bits, 4–73
division by zero, 4–75
DZED with DZE arithmetic trap, 4–72
DZED with INV arithmetic trap, 4–72
IEEE compliance and, B–2
inexact result, 4–74
invalid operation, 4–75
overflow disable, 4–75
underflow, 4–74
underflow to zero, 4–74
when unimplemented, 4–73
Trap enable bits, B–4
Trap handler, with non-finite arithmetic operands, 4–70
Trap handling, IEEE floating-point, B–6

Index–12
Trap modes
- floating-point, 4-68
- IEEE, 4-69
- IEEE convert-to-integer, 4-69
- VAX, 4-68
- VAX convert-to-integer, 4-69

Trap shadow
- defined, 4-69
- defined for floating-point, 4-65
- programming implications for, 5-30
- trap handler requirement for, 4-71

True result, 4-65
True zero, 4-65
T_floating data type
- alignment of, 2-10
- exceptions, 2-10
- format, 2-9
- MAX/MIN, 4-66
- NaN with S_floating convert, 4-82

U

UMULH instruction, 4-37
- with MULQ, 4-36
UNALIGNED data objects, 1-9
Unconditional long jump, 4-24
UNDEFINED operations, 1-7
Underflow enable (UNFE)
- FP_C quadword bit, B-5
Underflow status (UNFS)
- FP_C quadword bit, B-5
UNDZ bit
- See Trap disable bits, underflow to zero
UNF bit
- See arithmetic traps, underflow
UNFD bit
- See Trap disable bits, underflow
UNOP code form, A-13
UNORDERED memory references, 5-10
Unpack to bytes instructions, 4-156
UNPKBL (Unpack bytes to longwords) instruction, 4-156
UNPKBW (Unpack bytes to words) instruction, 4-156
UNPREDICTABLE results, 1-7
/U opcode qualifier
- IEEE floating-point, 4-69
- VAX floating-point, 4-68
Updated datum, 5-6
urti (PALcode) instruction, 10-1

V

VAX compatibility instructions, restrictions for, 4-149
VAX compatibility register, 3-3
VAX convert-to-integer trap mode, 4-69
VAX floating-point
- See also Floating-point instructions
  D_floating, 2-6
  F_floating, 2-3
  G_floating, 2-5
  trap modes, 4-69
VAX floating-point instructions
- add instructions, 4-104
- compare instructions, 4-106
- convert from integer instructions, 4-110
- convert to integer instructions, 4-109
- convert VAX floating format instructions, 4-111
- divide instructions, 4-117
- from integer move, 4-122
- function codes for, C-9
- function field format, 4-81
- multiply instructions, 4-124
- operate instructions, 4-96
- square root instructions, 4-126
- subtract instructions, 4-128
VAX rounding modes, 4-66
VAX trap modes, required instruction notation, 4-69
Vector instructions
- byte and word maximum, 4-152
- byte and word minimum, 4-152
Virtual D-cache, 5-4
- maintaining coherency of, 5-4
Virtual I-cache, 5-4
- maintaining coherency of, 5-5
Visibility, defined, 5-14
/V opcode qualifier
- IEEE floating-point, 4-69
- VAX floating-point, 4-69

W

WH64 (Write hint) instruction, 4-144
WH64 instruction
  lock_flag with, 4-10
whami (PALcode) instruction, 10-3
Windows NT Alpha PALcode, instruction summary, C-18
Windows NT Alpha privileged PALcode instructions, 11-2
Windows NT Alpha unprivileged PALcode instructions, 11-1

Index–13
WMB (Write memory barrier) instruction, 4–146
  atomic operations with, 5–8
  compared with MB, 4–147
  with shared data structures, 5–9
Word data type, 2–2
  atomic access of, 5–2
Word manipulation instructions
  LDWU, 4–6
  wrent (PALcode) instruction, 10–3
  wrentry (PALcode) instruction, 11–5
  wrfen (PALcode) instruction, 10–3
  wripr (PALcode) instruction, 10–3
  Write-back caches, requirements for, 5–5
  Write buffers, requirements for, 5–5
  WRITE_UNQ (PALcode) instruction, 9–6
  wrkgp (PALcode) instruction, 10–3
  wrmces (PALcode) instruction, 10–3, 11–5
  wrperfmon (PALcode) instruction, 10–3, 11–5
  wrunique (PALcode) instruction, 10–1, 11–5
  required recognition of, 6–5
  WRUNIQUE (PALcode) instruction
    required recognition of, 6–5
wrusp (PALcode) instruction, 10–4
wrval (PALcode) instruction, 10–4
wrvptptr (PALcode) instruction, 10–4
WR_PS_SW (PALcode) instruction, 9–6
wtint (PALcode) instruction, 10–4
WTINT (PALCODE) instruction, 9–8

X

x MOD y operator, 3–7
XOR instruction, 4–43
XOR operator, 3–8
X_floating data type, 2–10
  alignment of, 2–11
  big-endian format, 2–12
  format, 2–10
  MAX/MIN, 4–66

Z

ZAP instruction, 4–62
ZAPNOT instruction, 4–62
Zero byte instructions, 4–62
ZEXT(x) operator, 3–8
Technical Support and Ordering Information

Obtaining Technical Support

If you need technical support or help deciding which literature best meets your needs, call the Digital Semiconductor Information Line:

United States and Canada  1–800–332–2717
Outside North America   +1–508–628–4760

or visit the Digital Semiconductor World-Wide Web Internet site:
http://www.digital.com/info/semiconductor

Ordering Digital Semiconductor Products

To obtain a Digital Semiconductor Product Catalog, contact the Digital Semiconductor Information Line, above.

Ordering Digital Semiconductor Literature

For a complete list of the available Digital Semiconductor literature, contact the Digital Semiconductor Information Line or visit Digital Semiconductor's World-Wide Web Internet site.